DOI 10.5281/zenodo.3909718

Course Overview

How can we understand how the brain works? This course provides an introduction to in vivo neuroimaging in humans using functional magnetic resonance imaging (fMRI). The goal of the class is to introduce: (1) how the scanner generates data, (2) how psychological states can be probed in the scanner, and (3) how this data can be processed and analyzed. Students will be expected to analyze brain imaging data using the opensource Python programming language. We will be using several packages such as numpy, matplotlib, nibabel, nilearn, fmriprep, and nltools. This course will be useful for students working in neuroimaging labs, completing a neuroimaging thesis, or interested in pursuing graduate training in fields related to cognitive neuroscience.

Goals

1. Learn the basics of fMRI signals
2. Introduce standard data preprocessing techniques
3. Introduce the general linear model
4. Introduce advanced analysis techniques

Overview

This course is designed primarily around learning the basics of fMRI data analysis using the Python programming language. We will cover a lot of ground from introducing the Python programming language, to signal processing, to working with opensource packages from the Python Scientific Computing community. The format will be slightly different than the typical in-person version of the course.

- All course materials will be made available online in the format of a jupyter book, https://dartbrains.org/. Lectures will primarily be delivered via publicly available pre-recorded lectures available on youtube.
- Each course module will have a jupyter book tutorial and an accompanying assignment that will require completing some type of programming task or analysis. Students will work within small groups throughout the term on their assignments.
- Scheduled classes will primarily take the form of a Q&A where solutions to the assignments will be discussed and specific topics can be discussed in more detail.
- There will be a single exam, in which students will have to replicate an analysis of a published study using real data.
- For the final project, students will analyze an existing publicly available dataset to answer a new research question. Unfortunately, we will not be able to collect a new fMRI dataset as the Dartmouth Brain Imaging Center is currently closed for this term.

Questions

Please post any questions that arise from the material in this course on our Discourse Page

License for this book

Acknowledgements

Dartbrains was created by Luke Chang and supported by an NSF CAREER Award 1848370 and the Dartmouth Center for the Advancement of Learning. Our jupyterhub server was built and maintained by the Research Computing staff at Dartmouth. Special thanks to Arnold Song, William Hamblen, Christian Darabos, and John Hudson.

Instructors

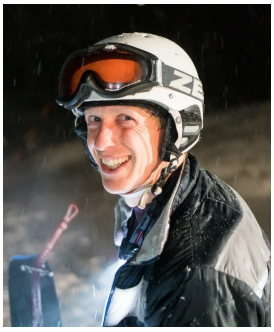
Professors

Luke Chang, PhD (Fall 2020) is an Assistant Professor of Psychological and Brain Sciences at Dartmouth College and directs the Computational Social Affective Neuroscience Laboratory. He completed a BA in psychology at Reed College, an MA in psychology at the New School for Social Research, and a PhD in clinical psychology and cognitive neuroscience at the University of Arizona with Alan Sanfey, PhD. He completed his predoctoral clinical internship training in behavioral medicine at the University of California Los Angeles and a postdoctoral fellowship at the University of Colorado Boulder under the mentorship of Tor Wager, PhD. His research program is focused on understanding the neurobiological and computational mechanisms underlying emotions and social interactions. Professor Chang is highly committed to innovating training in methods. He is the lead developer of the dartbrains course, the nltools python data analysis project, the Computational Social and Affective Neuroscience community page, and Co-Director of the Methods in Neuroscience at Dartmouth Computational Summer School.

Emily Finn, PhD (Winter 2021) is an Assistant Professor of Psychological and Brain Sciences at Dartmouth College and directs the Functional Imaging & Naturalistic Neuroscience (FINN) Lab. She completed a BA in linguistics at Yale University and a PhD in neuroscience, also at Yale. She then did her postdoctoral training in the Section on Functional Imaging Methods in the Laboratory of Brain and Cognition and the National Institute of Mental Health. Her research is focused on individual variability in brain activity and behavior, especially as it relates to appraisal of ambiguous information under naturalistic conditions. Professor Finn is committed to the ideals of open science, including data and code sharing (see examples here, here, and here, and to helping train other scientists in innovative new methods for neuroimaging data acquisition and analysis.

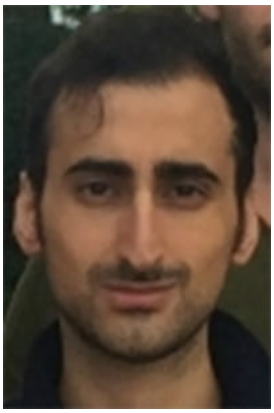
Tor Wager, PhD (Spring 2021) is the Diana L. Taylor Distinguished Professor in Neuroscience at Dartmouth College. He received his Ph.D. from the University of Michigan in Cognitive Psychology in 2003, and served as an Assistant (2004-2008) and Associate Professor (2009) at Columbia University, and as Associate (2010-2014) and Full Professor (2014-2019) at the University of Colorado, Boulder. Since 2004, he has directed the Cognitive and Affective Neuroscience laboratory, a research lab devoted to work on the neurophysiology of affective processes—pain, emotion, stress, and empathy—and how they are shaped by cognitive and social influences. Dr. Wager and his

lab are also dedicated to developing analysis methods for functional neuroimaging and sharing ideas, tools, and scientific data with the scientific community and public. See https://canlab.github.io for papers, data, tools, and code.



Jeremy Huckins, PhD (Fall 2019) is a Lecturer and Post-Doctoral researcher in the department of Psychological and Brain Sciences at Dartmouth College. He completed a BA in Neuroscience at Bowdoin College, worked with as a researcher with the King Lab at Harvard Medical School then completed a PhD in Experimental and Molecular Medicine at Dartmouth College. His current research program is focused on gaining insights into mental health using fMRI and mobile smartphone sensing.

Teaching Assistants



Sharif Saleki (Fall 2020) is a graduate student at Dartmouth college working with Peter Tse.

Bryan Gonzalez (Spring 2020) is a graduate student at Dartmouth college working with Luke Chang. He received his BA and MA from NYU. After a stint as a producer in creative media industries, his interest in research began in the Social Relations Lab at Columbia University studying speech mimicry. He later spent time learning polysomnography at Weill Cornell before coming to NYU Langone as a senior research coordinator. There, his efforts focused on finding biological markers of PTSD. At Dartmouth, Bryan is primarily interested in the computational mechanisms underpinning theory of mind across perceptual, behavioral and cognitive domains. His research is strongly influenced by reinforcement learning models, and probes mental state attribution in action understanding, preference learning, and anthropomorphism in human-robot interaction. Bryan is also passionate about promoting diversity in STEM education. In his free time, he loves running, live music, and curling up with his cat, “Puppy”.

Kirsten Ziman (Fall 2019) is a graduate student at Dartmouth college. She completed her bachelor's in Neuroscience at the University of Southern California, and worked in research settings at USC and UCLA before joining the Dartmouth community. Currently, she is studying attention and memory with professor Jeremy Manning.

Sasha Brietzke (Spring 2019) is graduate student at Dartmouth College. She completed a BA at Johns Hopkins University and an IRTA postbac fellowship at the NIH. She currently works in the Dartmouth Social Neuroscience Lab investigating the self through a social cognitive lens.

Syllabus

INSTRUCTORS

Professor: Luke Chang, PhD

Email: mailto:luke.j.chang@dartmouth.edu

Office Hours: Tuesday 1-2pm

Teaching Assistant: Sharif Saleki

Email: mailto:sharif.saleki.gr@dartmouth.edu

Office Hours: Monday & Wednesday, 11:30 - 12:30 pm

SPACE AND TIME

Remote Zoom Meetings 2A Tue/Thurs: 2:25-4:15 X-Hour Wed: 4:35-5:25

CLASS FORMAT

The format of Psych60 will be slightly different than the typical in-person version of the course. All course materials will be made available online in the format of a jupyter book, https://dartbrains.org/. Lectures will primarily be delivered via publicly available pre-recorded lectures available on youtube. Each course module will have a jupyter book tutorial and an accompanying assignment that will require completing some type of programming task or analysis. Students will work within small groups throughout the term on their assignments. Scheduled classes will primarily take the form of a Q&A where solutions to the assignments will be discussed and specific topics can be discussed in more detail. There will be a single exam, in which students will have to replicate an analysis of a published study using real data. For the final project, students will analyze an existing publicly available dataset to answer a new research question. Unfortunately, we will not be able to collect a new fMRI dataset as the Dartmouth Brain Imaging Center is currently closed for this term.

ZOOM MEETINGS

All class meetings will be held on zoom. By participating in the zoom meetings, all students will consent to the following:

** (1) Consent to recording of course and group office hours**

- I affirm my understanding that this course and any associated group meetings involving students and the instructor, including but not limited to scheduled and ad hoc office hours and other consultations, may be recorded within any digital platform used to offer remote instruction for this course;
- I further affirm that the instructor owns the copyright to their instructional materials, of which these recordings constitute a part, and distribution of any of these recordings in whole or in part without prior written consent of the instructor may be subject to discipline by Dartmouth up to and including expulsion;
- I authorize Dartmouth and anyone acting on behalf of Dartmouth to record my participation and appearance in any medium, and to use my name, likeness, and voice in connection with such recording; and
- I authorize Dartmouth and anyone acting on behalf of Dartmouth to use, reproduce, or distribute such recording without restrictions or limitation for any educational purpose deemed appropriate by Dartmouth and anyone acting on behalf of Dartmouth.

** (2) Requirement of consent to one-on-one recordings**

- By enrolling in this course, I hereby affirm that I will not under any circumstance make a recording in any medium of any one-on-one meeting with the instructor without obtaining the prior written consent of all those participating, and I understand that if I violate this prohibition, I will be subject to discipline by Dartmouth up to and including expulsion, as well as any other civil or criminal penalties under applicable law.

COMMUNICATION

For this class, we will have most of our discussion through the dart-psych60.slack.com channel.

TEXTBOOK

[OPTIONAL] Lindquist, M. & Wager, T (2015). Principles of fMRI. Available from https://leanpub.com/principlesoffmri.

ONLINE VIDEOS

Students are encouraged to watch assigned videos freely available online to supplement the classroom experience. Most videos will be available on youtube from:

- Principles of fMRI Course by Tor Wager & Martin Lindquist
- Analyzing Neural Time Series Data: Principles & Theory By Mike X Cohen
- Mumford Brain Stats by Jeanette Mumford

LECTURES

Lectures will primarily be delivered via freely available youtube videos. Students are expected to watch these on their own time and bring question to class or post on slack.

READINGS

Readings will supplement the online lectures and will be made available via Canvas.

CLASS PARTICIPATION (15% of grade)

You will be expected to participate in class discussions each day of class. This might include asking clarifying questions or helping another student.

HOMEWORK (30% of grade)

We will have occasional homework assignments based on lab assignments. You will be required to upload your jupyter notebook to the canvas site by midnight the day the homework is due.

ANATOMY FLASH (5% of grade)

In each class, a student will present a very brief presentation on one region of the brain. Flash presentations will include how to identify the region anatomically, a quick overview of its function, and a brief example of an interesting imaging study that identified a functional property of the region. Presentations should be developed on google slides and should be between 2-5 minutes.

EXAM (20% of grade)

To ensure that students learn key concepts about the principles of fMRI data analysis, we will have one exam that will involve analyzing a new dataset to demonstrate competence of analysis skills.

ANALYSIS PROJECT (30% of grade)

A key component of this course is learning how to process and analyze imaging data. Students will be introduced to key concepts during the laboratory assignments. Students will be expected to apply what they learn to analyzing an fMRI dataset. Most students will likely analyze the data collected in class, but are free to analyze any publicly available dataset (e.g., https://openfmri.org/, https://www.datalad.org/datasets.html). Students may work in small groups (~2-3 people), but each will independently write a final report of the research. The final written paper should be in journal format using APA style with an abstract, intro, methods, results, discussion, and references. Format 12-20 typed double-space pages, 11pt Ariel or Times font. (bibliography not included in the page limit). Each person is expected to write their own intro and conclusion, but the group can collaborate on the methods and results sections if they want. At the end of the class each group will give a ~10 minute presentation on their project (background, hypothesis, experimental design, results, analyses, conclusions). Paper Due at Midnight on the last day of class.

CLASSROOM POLICIES

HONOR CODE

Students are expected to strictly adhere to the Dartmouth Academic Honor Principle. As described in the Student Handbook, fundamental to the principle of independent learning is the requirement of honesty and integrity in the performance of academic assignments, both in the classroom and outside. Dartmouth operates on the principle of academic honor. Students who submit work that is not their own or who commit other acts of academic dishonesty will forfeit the opportunity to continue at Dartmouth. If you have questions or concerns regarding this policy during the course, please contact Professor Chang.

PLAGIARISM

Writing about scientific publications without just rephrasing is difficult, particularly when not everything is fully understood. Doing this properly takes time and practice, and one goal of the course is to move us in that direction. I don't expect to see a perfect scientific treatment at this stage. But I do want to see evidence of independent thought when considering the material and implications (rather than just regurgitating it), and some degree of creativity. When quoting, be sure appropriate citations are made.

MISSED ASSIGNMENTS

A student will only be excused from an assignment by permission of the Instructor and on the basis of a written note from a dean, doctor, or supervisor of official college-sponsored events being held off-campus and requiring a students' absence. If excused, a make-up must be taken as soon as possible (usually within 1 day of the originally-scheduled exam/assignment date).

LATE ASSIGNMENTS

All papers and presentations are due at the date and time specified. Scores for late papers will be reduced by 10% for every 24-hour period a paper is late. No extensions will be granted due to computer failure, roommate difficulties, printing problems, etc. According to College policy, there are no excused absences from class for

participation in College-sponsored extracurricular activities.

DISABILITIES

Any student with a documented disability needing academic adjustments or accommodations is requested to speak with me by the end of the second week of the term. All discussions will remain confidential, although the Academic Skills Center may be consulted to verify the documentation of the disability.

MENTAL HEALTH

The academic environment at Dartmouth is challenging, our terms are intensive, and classes are not the only demanding part of your life. There are a number of resources available to you on campus to support your wellness, including your undergraduate dean, Counseling and Human Development, and the Student Wellness Center. I encourage you to use these resources to take care of yourself throughout the term, and email me if you experience any difficulties.

RELIGIOUS OBSERVANCES

Some students may wish to take part in religious observances that occur during this academic term. If you have a religious observance which conflicts with your participation in the course, please meet with me by the end of the second week of the term to discuss appropriate accommodations.

TITLE IX

At Dartmouth, we value integrity, responsibility, and respect for the rights and interests of others, all central to our Principles of Community. We are dedicated to establishing and maintaining a safe and inclusive campus where all have equal access to the educational and employment opportunities Dartmouth offers. We strive to promote an environment of sexual respect, safety, and well-being. In its policies and standards, Dartmouth demonstrates unequivocally that sexual assault, gender-based harassment, domestic violence, dating violence, and stalking are not tolerated in our community.

The Sexual Respect Website at Dartmouth provides a wealth of information on your rights with regard to sexual respect and resources that are available to all in our community.

Please note that, as a faculty member, I am obligated to share disclosures regarding conduct under Title IX with Dartmouth's Title IX Coordinator. Confidential resources are also available, and include licensed medical or counseling professionals (e.g., a licensed psychologist), staff members of organizations recognized as rape crisis centers under state law (such as WISE), and ordained clergy (see https://dartgo.org/titleix_resources).

Should you have any questions, please feel free to contact Dartmouth's Title IX Coordinator or the Deputy Title IX Coordinator for the Guarini School. Their contact information can be found on the sexual respect website at: https://sexual-respect.dartmouth.edu.

Schedule

Measurement and Signal

- Readings: Huettel, ch 7
- Videos: Principles of fMRI Modules 5-8
- Lab: Introduction to Python
- Lab: Introduction to Data Frames & Plotting

Image Processing

- Readings: Poldrack ch 1,2
- Videos: Principles of fMRI Modules 9
- Lab: Introduction to Neuroimaging Data
- Lab: Signal vs Noise with ICA

Signal Processing

- Readings: Cohen ch 10, 11
- Videos: Cohen lecturelets

- Lab: Signal Processing Basics

Data Preprocessing

- Readings: Poldrack ch 3,4
- Videos: Principles of fMRI Modules 13-14
- Lab: Preprocessing with Nipype Quickstart
- Lab: Building Preprocessing Worklows with Nipype
- Lab: Automated Preprocessing with fMRIprep

General Linear Model

- Readings: Poldrack ch 5
- Videos: Principles of fMRI Modules 15-22
- Lab: Introduction to the General Linear Model
- Lab: Modeling Single Subject Data

Group Analysis

- Readings: Poldrack ch 6
- Videos: Principles of fMRI Modules 23-25
- Lab:

Multiple Comparisons

- Readings: Poldrack ch 7
- Videos: Principles of fMRI Modules 26-29
- Lab:

Connectivity

- Readings: Poldrack ch 8
- Videos:
- Lab:

Prediction/Classification

- Readings: Poldrack ch 9
- Videos:
- Lab:

Representational Similarity Analysis

- Readings: Haxby et al., 2014; Kriegeskorte et al., 2008
- Videos:
- Lab:

Intersubject Synchrony

- Readings: Hasson et al., 2011; Nummenmaa et al, 2018
- Videos:
- Lab:

Introduction to JupyterHub

Written by Luke Chang & Jeremy Huckins

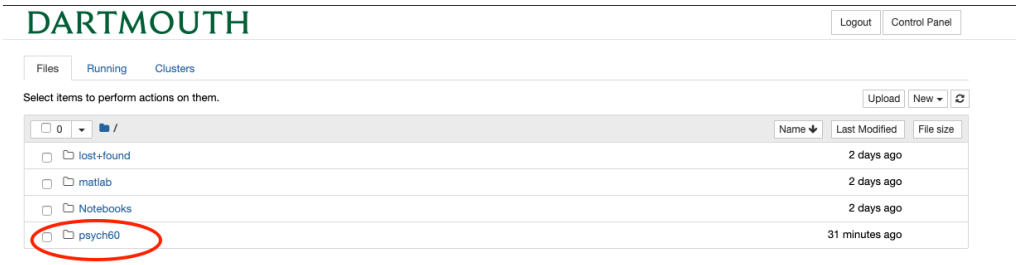
In this course we will primarily be using python to learn about fMRI data analysis. All of the laboratories can be run on your own individual laptops once you have installed Python (preferably via an anaconda distribution. However, the datasets are large and there can be annoying issues with different versions of packages and installing software across different operating systems. We will also occasionally be using additional software that will be called by Python (e.g., preprocessing). We have a docker container available that will contain all of the software and have created tutorials to download the data. In addition, some of the analyses we will run can be very computationally expensive and may exceed the capabilities of your laptop.

To meet these needs, Dartmouth's Research Computing has generously provided a dedicated server hosted on Amazon Web Services that will allow us to store data, access specialized software, and run analyses. This means that everyone should be able to run all of the tutorials on their laptops, tablets, etc by accessing notebooks on the jupyterhub server and will not need to install anything beyond a working browser.

Login

The main portal to access this resource will be through the Jupyterhub interface. This allows you to remotely login in to the server through your browser at https://jhub.dartmouth.edu using your netid. Please let us know if you are having difficulty logging in.

Once you've logged in you should see a screen like this.



The `Psych60` folder contains all of the relevant notebooks and data for the course.

Every time you login jupyterhub will spin up a new server just for you and will update all of the files.

Server

Every student will be able to have their own personal server to work on. This server is running on AWS cloud computing and should have all of the software you need to run the tutorials. If your server is idle for 10 minutes, it will automatically shut down. There are also a limited amount of resources available (e.g., storage, RAM). Each user has access to 512mb of RAM, keep an eye on how much your jobs are using. The server may crash if it exceeds 512mb.

Jupyter Notebooks

Jupyter notebooks are a great way to have your code, comments and results show up inline in a web browser. Work for this class will be done in Jupyter notebooks so you can reference what you have done, see the results and someone else could redo it in the future, similar to a typical lab notebook.

Rather than writing and re-writing an entire program, you can write lines of code and run them one at a time. Then, if you need to make a change, you can go back and make your edit and rerun the program again, all in the same window. In our specific case, we are going to use JupyterHub which lets several people access the same computer and data at the same time through a web browser.

Finally, you can view examples and share your work with the world very easily through nbviewer. One easy trick if you use a cloud storage service like dropbox is to paste a link to the dropbox file in nbviewer. These links will persist as long as the file remains being shared via dropbox.

Do not work directly on the notebooks in the `Psych60/notebook` folder. These will always be updating as I edit them. Instead, make sure you copy the notebooks you are working on to **your own personal folder**. These can only be changed by you and won't be deleted or updated when your server starts.

Opening a notebook on the server

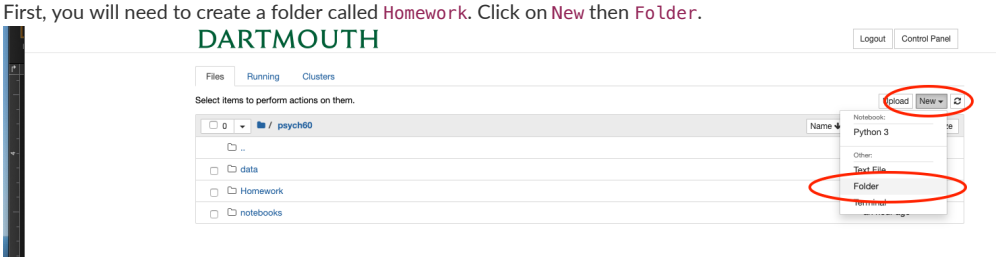
Click on Files, then Psych60, then notebooks. Click on any notebook you would like to load. Make sure you copy the notebook to another location outside of Psych60 to make sure your work won't be deleted.

For example, our first laboratory will be **1_Introduction_to_Programming.ipynb**.

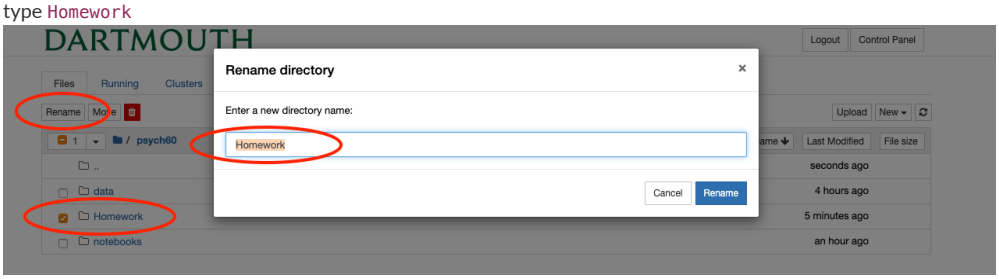
Copying Notebook

Make sure you copy your notebook to a different directory to make sure it will not be erased when you restart your server.

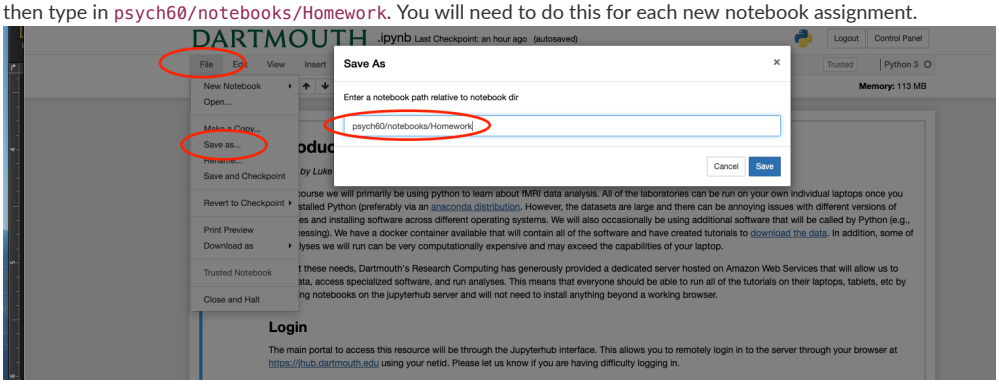
First, you will need to create a folder called `Homework`. Click on `New` then `Folder`.



Second, you will need to rename folder. `Check` the box next to the new untitled folder, then click `Rename`, then type `Homework`



Third, for all notebooks you will need to save a copy into your homework directory. Go to `File` then `Save as` then type in `psych60/notebooks/Homework`. You will need to do this for each new notebook assignment.



Alternative to Jupyterhub

If you use jupyter notebooks on your own computer then you own computer will be doing the processing. If you put your computer to sleep then processing will stop. It will also likely slow down other programs you are using on your computer. I would recommend installing it on your own computer so you can learn more about how to use it, or if you are interested in tinkering with the software or you happen to have a particularly fast/newer computer. We don't recommend going this route unless you don't have reliable access to the internet.

Please contact Professor Chang if you want any assistance doing this.

Installing Jupyter Notebooks on your own computer

1. Install python. We recommend using the Acaconda Distribution as it comes with most of the relevant scientific computing packages we will be using. Be sure to download Python 3.

Alternative 1: Install jupyter notebook (it comes with Anaconda)

```
pip install jupyter
```

Alternative 2: If you already have python installed:

```
pip install --upgrade pip
```

```
pip install jupyter
```

Starting Jupter Notebooks on your computer

Open a terminal, navigate to the directory you want to work from then type `jupyter notebook` or `jupyter lab`

Plotting and Atlases

For most of our labs we will be using Python to plot our data and results. However, it is often useful to have a more interactive experience. We recommend additionally downloading FSLeyes, which is a standalone image viewer developed by FSL. It can be installed by either downloading directly from the website, or using `pip`.

```
pip install fsleyes
```

If you are using a mac, you will likely also need to add an X11 window system such as xQuartz for the viewer to work properly.

References

Jupyter Dashboard Walkthrough

Jupyter Notebook Manual

Getting Started With Jupyter Notebook

Markdown Cheatsheet

Convert jupyter notebook to slides

Download Data

Written by Luke Chang

Many of the imaging tutorials throughout this course will use open data from the Pinel Localizer task.

The Pinel Localizer task was designed to probe several different types of basic cognitive processes, such as visual perception, finger tapping, language, and math. Several of the tasks are cued by reading text on the screen (i.e., visual modality) and also by hearing auditory instructions (i.e., auditory modality). The trials are randomized across conditions and have been optimized to maximize efficiency for a rapid event related design. There are 100 trials in total over a 5-minute scanning session. Read the original paper for more specific details about the task and the dataset paper.

This dataset is well suited for these tutorials as it is (a) publicly available to anyone in the world, (b) relatively small (only about 5min), and (c) provides many options to create different types of contrasts.

There are a total of 94 subjects available, but we will primarily only be working with a smaller subset of about 30.

Downloading the data is very easy as it is currently available on the OSF website and also

We will use the `osfclient` package to download the entire dataset. Note, that the entire dataset is fairly large (~5.25gb), so make sure you have space on your computer. At some point, we will make a smaller version for the dartbrain course available for download.

If you are taking the Psych60 course at Dartmouth, we have already made the download available on the jupyterhub server.

Let's first make sure the `osfclient` package is installed in our python environment.

In this notebook, we will walk through how to access the datset using DataLad.

DataLad

The easist way to access the data is using DataLad, which is an open source version control system for data built on top of git-annex. Think of it like git for data. It provides a handy command line interface for downloading data, tracking changes, and sharing it with others.

While DataLad offers a number of useful features for working with datasets, there are three in particular that we think make it worth the effort to install for this course.

1. Cloning a DataLad Repository can be completed with a single line of code `datalad clone <repository>` and provides the full directory structure in the form of symbolic links. This allows you to explore all of the files in the dataset, without having to download the entire dataset at once.
2. Specific files can be easily downloaded using `datalad get <filename>`, and files can be removed from your computer at any time using `datalad drop <filename>`. As these datasets are large, this will allow you to only work with the data that you need for a specific tutorial and you can drop the rest when you are done with it.
3. All of the DataLad commands can be run within Python using the datalad python api.

We will only be covering a few basic DataLad functions to get and drop data. We encourage the interested reader to read the very comprehensive DataLad User Handbook for more details and troubleshooting.

Installing Datalad

DataLad can be easily installed using pip.

```
pip install datalad
```

Unfortunately, it currently requires manually installing the git-annex dependency, which is not automatically installed using pip.

If you are using OSX, we recommend installing git-annex using homebrew package manager.

```
brew install git-annex
```

If you are on Debian/Ubuntu we recommend enabling the NeuroDebian repository and installing with apt-get.

```
sudo apt-get install datalad
```

For more installation options, we recommend reading the DataLad installation instructions.

```
!pip install datalad
```

```
Requirement already satisfied: datalad in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (0.12.6)
Requirement already satisfied: msgpack in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (1.0.0)
Requirement already satisfied: appdirs in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (1.4.3)
Requirement already satisfied: chardet>=3.0.4 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (3.0.4)
Requirement already satisfied: keyring>=8.0 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (21.4.0)
Requirement already satisfied: GitPython>=2.1.12 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (3.1.0)
Requirement already satisfied: fasteners in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (0.15)
Requirement already satisfied: jsmin in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (2.2.2)
Requirement already satisfied: iso8601 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (0.1.12)
Requirement already satisfied: keyrings.alt in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (3.4.0)
Requirement already satisfied: patool>=1.7 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (1.12)
Requirement already satisfied: wrapt in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (1.11.2)
Requirement already satisfied: tqdm in /Users/lukechang/anaconda3/lib/python3.7/site-
packages (from datalad) (4.48.2)
Requirement already satisfied: whoosh in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (2.7.4)
Requirement already satisfied: boto in /Users/lukechang/anaconda3/lib/python3.7/site-
packages (from datalad) (2.49.0)
Requirement already satisfied: simplejson in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (3.17.0)
Requirement already satisfied: PyGithub in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (1.47)
Requirement already satisfied: humanize in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (2.4.0)
Requirement already satisfied: requests>=1.2 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from datalad) (2.24.0)
Requirement already satisfied: importlib-metadata; python_version < "3.8" in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from keyring>=8.0->datalad)
(1.7.0)
Requirement already satisfied: gitdb<5,>=4.0.1 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from GitPython>=2.1.12-
>datalad) (4.0.2)
Requirement already satisfied: monotonic>=0.1 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from fasteners->datalad)
(1.5)
Requirement already satisfied: six in /Users/lukechang/anaconda3/lib/python3.7/site-
packages (from fasteners->datalad) (1.15.0)
Requirement already satisfied: pyjwt in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from PyGithub->datalad)
(1.7.1)
Requirement already satisfied: deprecated in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from PyGithub->datalad)
(1.2.9)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from requests>=1.2->datalad)
(1.25.10)
Requirement already satisfied: certifi>=2017.4.17 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from requests>=1.2->datalad)
(2020.6.20)
Requirement already satisfied: idna<3,>=2.5 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from requests>=1.2->datalad)
(2.10)
Requirement already satisfied: zipp>=0.5 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from importlib-metadata;
python_version < "3.8"->keyring>=8.0->datalad) (3.1.0)
Requirement already satisfied: smmap<4,>=3.0.1 in
/Users/lukechang/anaconda3/lib/python3.7/site-packages (from gitdb<5,>=4.0.1-
>GitPython>=2.1.12->datalad) (3.0.1)
```

Download Data with DataLad

The Pinel localizer dataset can be accessed at the following location https://gin.g-node.org/ljchang/Localizer/. To download the Localizer dataset run `datalad install https://gin.g-node.org/ljchang/Localizer` in a terminal in the location where you would like to install the dataset. Don't forget to change the directory to a folder on your local computer. The full dataset is approximately 42gb.

You can run this from the notebook using the `!` cell magic.

```
%cd ~/Dropbox/Dartbrains/data

!datalad install https://gin.g-node.org/ljchang/Localizer
```

```
/Users/lukechang/Dropbox/Dartbrains/data
```

Datalad Basics

You might be surprised to find that after cloning the dataset that it barely takes up any space `du -sh`. This is because cloning only downloads the metadata of the dataset to see what files are included.

You can check to see how big the entire dataset would be if you downloaded everything using `datalad status`.

```
%cd ~/Dropbox/Dartbrains/data/Localizer

!datalad status --annex
```

```
/Users/lukechang/Dropbox/Dartbrains/data/Localizer
1794 annex'd files (42.1 GB recorded total size)
```

Getting Data

One of the really nice features of datalad is that you can see all of the data without actually storing it on your computer. When you want a specific file you use `datalad get <filename>` to download that specific file. Importantly, you do not need to download all of the dat at once, only when you need it.

Now that we have cloned the repository we can grab individual files. For example, suppose we wanted to grab the first subject's confound regressors generated by fmriprep.

```
!datalad get participants.tsv
```

```

```

Now we can check and see how much of the total dataset we have downloaded using `datalad status`

```
!datalad status --annex all
```

```
1794 annex'd files (0.0 B/42.1 GB present/total size)
```

If you would like to download all of the files you can use `datalad get .`. Depending on the size of the dataset and the speed of your internet connection, this might take awhile. One really nice thing about datalad is that if your connection is interrupted you can simply run `datalad get .` again, and it will resume where it left off.

You can also install the dataset and download all of the files with a single command `datalad install -g https://gin.g-node.org/ljchang/Localizer`. You may want to do this if you have a lot of storage available and a fast internet connection. For most people, we recommend only downloading the files you need for a specific tutorial.

Dropping Data

Most people do not have unlimited space on their hard drives and are constantly looking for ways to free up space when they are no longer actively working with files. Any file in a dataset can be removed using `datalad drop`. Importantly, this does not delete the file, but rather removes it from your computer. You will still be able to see file metadata after it has been dropped in case you want to download it again in the future.

As an example, let's drop the Localizer participants .tsv file.

```
!datalad drop participants.tsv
```

Datalad has a Python API!

One particularly nice aspect of datalad is that it has a Python API, which means that anything you would like to do with datalad in the commandline, can also be run in Python. See the details of the datalad Python API.

For example, suppose you would like to clone a data repository, such as the Localizer dataset. You can run `dl.clone(source=url, path=location)`. Make sure you set `localizer_path` to the location where you would like the Localizer repository installed.

```
import os
import glob
import datalad.api as dl
import pandas as pd

localizer_path = '/Users/lukechang/Dropbox/Dartbrains/data/Localizer'

dl.clone(source='https://gin.g-node.org/ljchang/Localizer', path=localizer_path)
```

```
[WARNING] realpath of PWD=/ is / whenever
os.getcwd()=/Users/lukechang/Dropbox/Dartbrains/data/Localizer. From now on will be
returning os.getcwd(). Directory symlinks in the paths will be resolved
```

```
<Dataset path=/Users/lukechang/Dropbox/Dartbrains/data/Localizer>
```

We can now create a dataset instance using `dl.Dataset(path_to_data)`.

```
ds = dl.Dataset(localizer_path)
```

How much of the dataset have we downloaded? We can check the status of the annex using `ds.status(annex='all')`.

```
results = ds.status(annex='all')
```

```
1794 annex'd files (0.0 B/42.1 GB present/total size)
1794 annex'd files (0.0 B/42.1 GB present/total size)
```

Looks like it's empty, which makes sense since we only cloned the dataset.

Now we need to get some data. Let's start with something small to play with first.

Let's use `glob` to find all of the tab-delimited confound data generated by fmriprep.

```
file_list = glob.glob(os.path.join(localizer_path, '*', 'fmriprep', '*', 'func',
'*tsv'))
file_list.sort()
file_list[:10]
```

```
['/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01/func/sub-S01_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S02/func/sub-S02_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S03/func/sub-S03_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S04/func/sub-S04_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S05/func/sub-S05_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S06/func/sub-S06_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S07/func/sub-S07_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S08/func/sub-S08_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S09/func/sub-S09_task-localizer_desc-confounds_regressors.tsv',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S10/func/sub-S10_task-localizer_desc-confounds_regressors.tsv']
```

glob can search the filetree and see all of the relevant data even though none of it has been downloaded yet.

Let's now download the first subjects confound regressor file and load it using pandas.

```
result = ds.get(file_list[0])

confounds = pd.read_csv(file_list[0], sep='\t')
confounds.head()
```

	csf	csf_derivative1	csf_derivative1_power2	csf_power2	white_matter	white_
0	5164.630182	NaN	NaN	2.667340e+07	4006.007667	
1	5178.481411	13.851229	191.856548	2.681667e+07	4011.819383	
2	5161.040643	-17.440768	304.180395	2.663634e+07	4006.766409	
3	5150.604178	-10.436465	108.919794	2.652872e+07	4008.586021	
4	5172.441161	21.836983	476.853810	2.675415e+07	4007.189291	

5 rows × 136 columns

What if we wanted to drop that file? Just like the CLI, we can use `ds.drop(file_name)`.

```
result = ds.drop(file_list[0])
```

To confirm that it is actually removed, let's try to load it again with pandas.

```
confounds = pd.read_csv(file_list[0], sep='\t')
```

Looks like it was successfully removed.

We can also load the entire dataset in one command if want using `ds.get(dataset='.', recursive=True)`. We are not going to do it right now as this will take awhile and require lots of free hard disk space.

Let's actually download one of the files we will be using in the tutorial. First, let's use glob to get a list of all of the functional data that has been preprocessed by fmriprep, denoised, and smoothed.

```
file_list = glob.glob(os.path.join(localizer_path, 'derivatives', 'fmriprep', '*',
'func', '*task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz'))
file_list.sort()
file_list
```

```
['/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01/func/sub-S01_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S02/func/sub-S02_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S03/func/sub-S03_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S04/func/sub-S04_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S05/func/sub-S05_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S06/func/sub-S06_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S07/func/sub-S07_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S08/func/sub-S08_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S09/func/sub-S09_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S10/func/sub-S10_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S11/func/sub-S11_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S12/func/sub-S12_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S13/func/sub-S13_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S14/func/sub-S14_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S15/func/sub-S15_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S16/func/sub-S16_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S17/func/sub-S17_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S18/func/sub-S18_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S19/func/sub-S19_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S20/func/sub-S20_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S21/func/sub-S21_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S22/func/sub-S22_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S23/func/sub-S23_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S24/func/sub-S24_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S25/func/sub-S25_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S26/func/sub-S26_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S27/func/sub-S27_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S28/func/sub-S28_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S29/func/sub-S29_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S30/func/sub-S30_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S31/func/sub-S31_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S32/func/sub-S32_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S33/func/sub-S33_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S34/func/sub-S34_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S35/func/sub-S35_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S36/func/sub-S36_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S37/func/sub-S37_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S38/func/sub-S38_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S39/func/sub-S39_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S40/func/sub-S40_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S41/func/sub-S41_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S42/func/sub-S42_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S43/func/sub-S43_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
```

```
S44/func/sub-S44_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S45/func/sub-S45_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S46/func/sub-S46_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S47/func/sub-S47_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S48/func/sub-S48_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S49/func/sub-S49_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S50/func/sub-S50_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S51/func/sub-S51_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S52/func/sub-S52_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S53/func/sub-S53_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S54/func/sub-S54_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S55/func/sub-S55_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S56/func/sub-S56_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S57/func/sub-S57_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S58/func/sub-S58_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S59/func/sub-S59_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S60/func/sub-S60_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S61/func/sub-S61_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S62/func/sub-S62_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S63/func/sub-S63_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S64/func/sub-S64_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S65/func/sub-S65_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S66/func/sub-S66_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S67/func/sub-S67_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S68/func/sub-S68_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S69/func/sub-S69_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S70/func/sub-S70_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S71/func/sub-S71_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S72/func/sub-S72_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S73/func/sub-S73_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S74/func/sub-S74_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S75/func/sub-S75_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S76/func/sub-S76_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S77/func/sub-S77_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S78/func/sub-S78_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S79/func/sub-S79_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S80/func/sub-S80_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S81/func/sub-S81_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S82/func/sub-S82_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S83/func/sub-S83_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S84/func/sub-S84_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S85/func/sub-S85_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S86/func/sub-S86_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S87/func/sub-S87_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
```

```
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S88/func/sub-S88_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S89/func/sub-S89_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S90/func/sub-S90_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S91/func/sub-S91_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S92/func/sub-S92_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S93/func/sub-S93_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S94/func/sub-S94_task-localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz']
```

Now let's download the first subject's file using `ds.get()`. This file is 825mb, so this might take a few minutes depending on your internet speed.

```
result = ds.get(file_list[0])
```

How much of the dataset have we downloaded? We can check the status of the annex using `ds.status(annex='all')`.

```
result = ds.status(annex='all')
```

```
1794 annex'd files (106.9 MB/42.1 GB present/total size)
1794 annex'd files (106.9 MB/42.1 GB present/total size)
```

Now let's download the preprocessed data for the first 15 subjects including the fmriprep reports.

```
file_list = glob.glob(os.path.join(localizer_path, '*', 'fmriprep', 'sub*'))
file_list.sort()
for f in file_list[:30]:
    result = ds.get(f)
```

```
[WARNING] Running get resulted in stderr output: git-annex: get: 1 failed

[ERROR] download failed: ResponseBodyTooShort 102374852 11803033
| download failed: Internal Server Error
| download failed: ResponseBodyTooShort 102374852 54798217
| download failed: Internal Server Error
| download failed: Terminated False "Error_Packet \"partial packet: expecting 4433
bytes, got: 3051\"" (Error_Packet "partial packet: expecting 4433 bytes, got: 3051")
| download failed: Internal Server Error
[get(/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01/anat/sub-S01_from-T1w_to-MNI152NLin2009cAsym_mode-image_xfm.h5)]
[WARNING] could not get some content in
/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-S01
['/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01/anat/sub-S01_from-T1w_to-MNI152NLin2009cAsym_mode-image_xfm.h5']
[get(/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01)]
```

```
---------------------------------------------------------------------------
IncompleteResultsError                    Traceback (most recent call last)
<ipython-input-13-a368202ed3ba> in <module>
      2 file_list.sort()
      3 for f in file_list[:30]:
----> 4     result = ds.get(f)

~/anaconda3/lib/python3.7/site-packages/datalad/distribution/dataset.py in
apply_func(wrapped, instance, args, kwargs)
    498             elif i >= ds_index:
    499                 kwargs[orig_pos[i+1]] = args[i]
--> 500         return f(**kwargs)
    501
    502     setattr(Dataset, name, apply_func(f))

~/anaconda3/lib/python3.7/site-packages/datalad/interface/utils.py in
eval_func(wrapped, instance, args, kwargs)
    490                     return results
    491             lgr.log(2, "Returning return_func from eval_func for %s",
wrapped_class)
--> 492             return return_func(generator_func)(*args, **kwargs)
    493
    494     return eval_func(func)

~/anaconda3/lib/python3.7/site-packages/datalad/interface/utils.py in
return_func(wrapped_, instance_, args_, kwargs_)
    478                     # unwind generator if there is one, this actually runs
    479                     # any processing
--> 480                     results = list(results)
    481                 # render summaries
    482                 if not result_xfm and result_renderer == 'tailored':

~/anaconda3/lib/python3.7/site-packages/datalad/interface/utils.py in
generator_func(*_args, **_kwargs)
    465                 raise IncompleteResultsError(
    466                     failed=incomplete_results,
--> 467                     msg="Command did not complete successfully")
    468
    469         if return_type == 'generator':

IncompleteResultsError: Command did not complete successfully [{'type': 'file',
'refds': '/Users/lukechang/Dropbox/Dartbrains/data/Localizer', 'status': 'error',
'path': '/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01/anat/sub-S01_from-T1w_to-MNI152NLin2009cAsym_mode-image_xfm.h5', 'action': 'get',
'annexkey': 'MD5E-s102374852--0e5fb08f5ddaef0db227511eb56f08fd.h5', 'message':
'download failed: ResponseBodyTooShort 102374852 11803033\ndownload failed: Internal
Server Error\ndownload failed: ResponseBodyTooShort 102374852 54798217\ndownload
failed: Internal Server Error\ndownload failed: Terminated False "Error_Packet
\\"partial packet: expecting 4433 bytes, got: 3051\\"" (Error_Packet "partial packet:
expecting 4433 bytes, got: 3051")\ndownload failed: Internal Server Error'},
{'action': 'get', 'path':
'/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-S01',
'type': 'directory', 'refds': '/Users/lukechang/Dropbox/Dartbrains/data/Localizer',
'status': 'impossible', 'message': ('could not get some content in %s %s',
'/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-S01',
['/Users/lukechang/Dropbox/Dartbrains/data/Localizer/derivatives/fmriprep/sub-
S01/anat/sub-S01_from-T1w_to-MNI152NLin2009cAsym_mode-image_xfm.h5'])}]
```

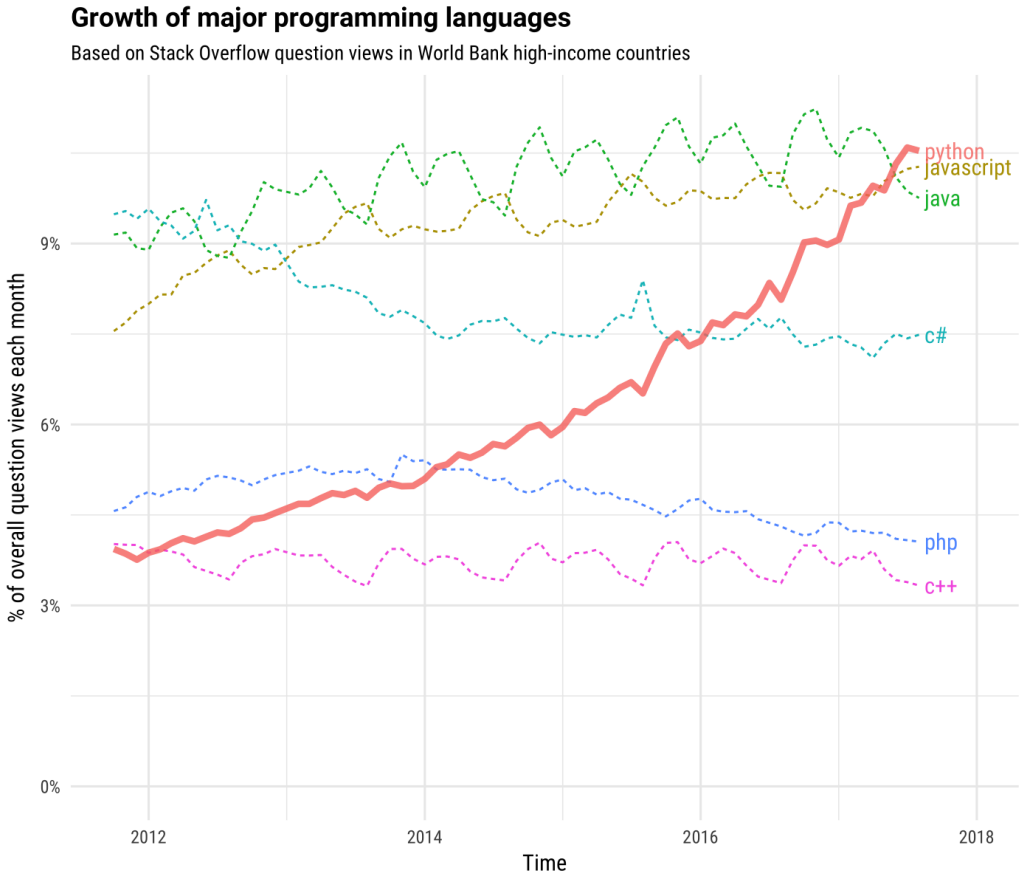
Ok, that concludes our tutorial for how to download data for this course with datalad using both the command line interface and also the Python API.

Introduction to programming

Written by Luke Chang

In this notebook we will begin to learn how to use Python. There are many different ways to install Python, but we recommend starting using Anaconda which is preconfigured for scientific computing. Start with installing Python 3.7. For those who prefer a more configurable IDE, Pycharm is a nice option. Python is a modular interpreted language with an intuitive minimal syntax that is quickly becoming one of the most popular languages for conducting research. You can use python for stimulus presentation, data analysis, machine-learning, scraping data, creating websites with flask or django, or neuroimaging data analysis.

There are lots of free useful resources to learn how to use python and various modules. See Jeremy Manning's or Yaroslav Halchenko's excellent Dartmouth courses. Codeacademy is a great interactive tutorial. Stack Overflow is an incredibly useful resource for asking specific questions and seeing responses to others that have been rated by the development community.



Jupyter Notebooks

We will primarily be using Jupyter Notebooks to interface with Python. A Jupyter notebook consists of **cells**. The two main types of cells you will use are code cells and markdown cells.

A ***code cell*** contains actual code that you want to run. You can specify a cell as a code cell using the pulldown menu in the toolbar in your Jupyter notebook. Otherwise, you can can hit esc and then y (denoted "esc, y") while a cell is selected to specify that it is a code cell. Note that you will have to hit enter after doing this to start editing it. If you want to execute the code in a code cell, hit "shift + enter." Note that code cells are executed in the order you execute them. That is to say, the ordering of the cells for which you hit "shift + enter" is the order in which the code is executed. If you did not explicitly execute a cell early in the document, its results are now known to the Python interpreter.

Markdown cells contain text. The text is written in markdown, a lightweight markup language. You can read about its syntax here. Note that you can also insert HTML into markdown cells, and this will be rendered properly. As you are typing the contents of these cells, the results appear as text. Hitting "shift + enter" renders the text in the formatting you specify. You can specify a cell as being a markdown cell in the Jupyter toolbar, or by hitting "esc, m" in the cell. Again, you have to hit enter after using the quick keys to bring the cell into edit mode.

In general, when you want to add a new cell, you can use the "Insert" pulldown menu from the Jupyter toolbar. The shortcut to insert a cell below is "esc, b" and to insert a cell above is "esc, a." Alternatively, you can execute a cell and automatically add a new one below it by hitting "alt + enter."

```
print("Hello World")
```

Package Management

Package managment in Python has been dramatically improving. Anaconda has it's own package manager called 'conda'. Use this if you would like to install a new module as it is optimized to work with anaconda.

```
!conda install *package*
```

However, sometimes conda doesn't have a particular package. In this case use the default python package manager called 'pip'.

These commands can be run in your unix terminal or you can send them to the shell from a Jupyter notebook by starting the line with `!`

It is easy to get help on how to use the package managers

```
!pip help install
```

```
!pip help install
```

```
!pip list --outdated
```

```
!pip install setuptools --upgrade
```

Variables

Python is a dynamically typed language, which means that you can easily change the datatype associated with a variable. There are several built-in datatypes that are good to be aware of.

- Built-in
  - Numeric types:
    - **int**, **float**, **long**, complex
  - String: **str**
  - Boolean: **bool**
    - True / False
  - **NoneType**
- User defined
- Use the type() function to find the type for a value or variable
- Data can be converted using cast commands

```
# Integer
a = 1
print(type(a))

# Float
b = 1.0
print(type(b))

# String
c = 'hello'
print(type(c))

# Boolean
d = True
print(type(d))

# None
e = None
print(type(e))

# Cast integer to string
print(type(str(a)))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'NoneType'>
<class 'str'>
```

Math Operators

- +, -, *, and /
- Exponentiation **
- Modulo %

- Note that division with integers in Python 2.7 automatically rounds, which may not be intended. It is recommended to import the division module from python3 `from __future__ import division`

```
# Addition
a = 2 + 7
print(a)

# Subtraction
b = a - 5
print(b)

# Multiplication
print(b*2)

# Exponentiation
print(b**2)

# Modulo
print(4%9)

# Division
print(4/9)
```

```
9
4
8
16
4
0.4444444444444444
```

String Operators

- Some of the arithmetic operators also have meaning for strings. E.g. for string concatenation use + sign
- String repetition: Use * sign with a number of repetitions

```
# Combine string
a = 'Hello'
b = 'World'
print(a + b)

# Repeat String
print(a*5)
```

```
HelloWorld
HelloHelloHelloHelloHello
```

Logical Operators

Perform logical comparison and return Boolean value

```
x == y # x is equal to y
x != y # x is not equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```

X	not X
True	False
False	True

X	Y	X AND Y	X OR Y
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

```
# Works for string
a = 'hello'
b = 'world'
c = 'Hello'
print(a==b)
print(a==c)
print(a!=b)

# Works for numeric
d = 5
e = 8
print(d < e)
```

```
False
False
True
True
```

Conditional Logic (if...)

Unlike most other languages, Python uses tab formatting rather than closing conditional statements (e.g., end).

- Syntax:

```
if condition:
    do something
```

- Implicit conversion of the value to bool() happens if `condition` is of a different type than **bool**, thus all of the following should work:

```
if condition:
    do_something
elif condition:
    do_alternative1
else:
    do_otherwise # often reserved to report an error
                 # after a long list of options
```

```
n = 1

if n:
    print("n is non-0")

if n is None:
    print("n is None")

if n is not None:
    print("n is not None")
```

```
n is non-0
n is not None
```

Loops

- **for** loop is probably the most popular loop construct in Python:

```
for target in sequence:
    do_statements
```

- However, it's also possible to use a **while** loop to repeat statements while `condition` remains True:

```
while condition do:
    do_statements
```

```
string = "Python is going to make conducting research easier"
for c in string:
    print(c)
```

```
P
y
t
h
o
n

i
s

g
o
i
n
g

t
o

m
a
k
e

c
o
n
d
u
c
t
i
n
g

r
e
s
e
a
r
c
h

e
a
s
i
e
r
```

```
x = 0
end = 10

csum = 0
while x < end:
    csum += x
    print(x, csum)
    x += 1
print(f"Exited with x=={x}")
```

```
0 0
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
Exited with x==10
```

Functions

A **function** is a named sequence of statements that performs a computation. You define the function by giving it a name, specify a sequence of statements, and optionally values to return. Later, you can "call" the function by name.

```
def make_upper_case(text):
    return (text.upper())
```

- The expression in the parenthesis is the **argument**.
- It is common to say that a function **"takes" an argument** and **"returns" a result**.
- The result is called the **return value**.

The first line of the function definition is called the **header**; the rest is called the **body**.

The header has to end with a colon and the body has to be indented. It is a common practice to use 4 spaces for indentation, and to avoid mixing with tabs.

Function body in Python ends whenever statement begins at the original level of indentation. There is no **end** or **fed** or any other identify to signal the end of function. Indentation is part of the the language syntax in Python, making it more readable and less cluttered.

```
def make_upper_case(text):
    return (text.upper())

string = "Python is going to make conducting research easier"

print(make_upper_case(string))
```

```
PYTHON IS GOING TO MAKE CONDUCTING RESEARCH EASIER
```

Python Containers

There are 4 main types of builtin containers for storing data in Python:

- list
- tuple
- dict
- set

Lists

In Python, a list is a mutable sequence of values. Mutable means that we can change separate entries within a list. For a more in depth tutorial on lists look here

- Each value in the list is an element or item
- Elements can be any Python data type
- Lists can mix data types
- Lists are initialized with `[]` or `list()`

```
l = [1,2,3]
```

-

Elements within a list are indexed (**starting with 0**)

```
l[0]
```

-

Elements can be nested lists

```
nested = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

-

Lists can be *sliced*.

```
l[start:stop:stride]
```

- Like all python containers, lists have many useful methods that can be applied

```
a.insert(index,new element)
a.append(element to add at end)
len(a)
```

-

List comprehension is a *Very* powerful technique allowing for efficient construction of new lists.

```
[a for a in l]
```

```
# Indexing and Slicing
a = ['lists','are','arrays']
print(a[0])
print(a[1:3])

# List methods
a.insert(2,'python')
a.append('.')
print(a)
print(len(a))

# List Comprehension
print([x.upper() for x in a])
```

```
lists
['are', 'arrays']
['lists', 'are', 'python', 'arrays', '.']
5
['LISTS', 'ARE', 'PYTHON', 'ARRAYS', '.']
```

Dictionaries

- In Python, a dictionary (or `dict`) is mapping between a set of indices (**keys**) and a set of **values**
- The items in a dictionary are key-value pairs
- Keys can be any Python data type
- Dictionaries are unordered
- Here is a more indepth tutorial on dictionaries

```
# Dictionaries
eng2sp = {}
eng2sp['one'] = 'uno'
print(eng2sp)

eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2sp)

print(eng2sp.keys())
print(eng2sp.values())
```

```
{'one': 'uno'}
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
dict_keys(['one', 'two', 'three'])
dict_values(['uno', 'dos', 'tres'])
```

Tuples

In Python, a **tuple** is an immutable sequence of values, meaning they can't be changed

- Each value in the tuple is an element or item
- Elements can be any Python data type
- Tuples can mix data types
- Elements can be nested tuples
- **Essentially tuples are immutable lists**

Here is a nice tutorial on tuples

```
numbers = (1, 2, 3, 4)
print(numbers)

t2 = 1, 2
print(t2)
```

```
(1, 2, 3, 4)
(1, 2)
```

sets

In Python, a `set` is an efficient storage for "membership" checking

- `set` is like a `dict` but only with keys and without values
- a `set` can also perform set operations (e.g., union intersection)
- Here is more info on sets

```
# Union
print({1, 2, 3, 'mom', 'dad'} | {2, 3, 10})

# Intersection
print({1, 2, 3, 'mom', 'dad'} & {2, 3, 10})

# Difference
print({1, 2, 3, 'mom', 'dad'} - {2, 3, 10})
```

```
{1, 2, 3, 'mom', 10, 'dad'}
{2, 3}
{1, 'mom', 'dad'}
```

Modules

A *Module* is a python file that contains a collection of related definitions. Python has *hundreds* of standard modules. These are organized into what is known as the Python Standard Library. You can also create and use your own modules. To use functionality from a module, you first have to import the entire module or parts of it into your namespace

- To import the entire module, use

```
import module_name
```

- You can also import a module using a specific name

```
import module_name as new_module_name
```

- To import specific definitions (e.g. functions, variables, etc) from the module into your local namespace, use

```
from module_name import name1, name2
```

which will make those available directly in your `namespace`

```
import os
from glob import glob
```

Here let's try and get the path of the current working directory using functions from the `os` module

```
os.path.abspath(os.path.curdir)
```

```
'/Users/lukechang/Dropbox/Dartbrains/Notebooks'
```

It looks like we are currently in the notebooks folder of the github repository. Let's use glob, a pattern matching function, to list all of the csv files in the Data folder.

```
data_file_list = glob(os.path.join('../..','Data','*csv'))
print(data_file_list)
```

```
[]
```

This gives us a list of the files including the relative path from the current directory. What if we wanted just the filenames? There are several different ways to do this. First, we can use the the `os.path.basename` function. We loop over every file, grab the base file name and then append it to a new list.

```
file_list = []
for f in data_file_list:
    file_list.append(os.path.basename(f))

print(file_list)
```

```
['salary_exercise.csv', 'salary.csv']
```

Alternatively, we could loop over all files and split on the `/` character. This will create a new list where each element is whatever characters are separated by the splitting character. We can then take the last element of each list.

```
file_list = []
for f in data_file_list:
    file_list.append(f.split('/')[-1])

print(file_list)
```

```
['salary_exercise.csv', 'salary.csv']
```

It is also sometimes even cleaner to do this as a list comprehension

```
[os.path.basename(x) for x in data_file_list]
```

```
['salary_exercise.csv', 'salary.csv']
```

Exercises

Find Even Numbers

Let's say I give you a list saved in a variable: a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]. Make a new list that has only the even elements of this list in it.

Find Maximal Range

Given an array length 1 or more of ints, return the difference between the largest and smallest values in the array.

Duplicated Numbers

Find the numbers in list a that are also in list b

a = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

b = [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]

Speeding Ticket Fine

You are driving a little too fast on the highway, and a police officer stops you. Write a function that takes the speed as an input and returns the fine.

If speed is 60 or less, the result is `$0`. If speed is between 61 and 80 inclusive, the result is `$100`. If speed is 81 or more, the result is `$500`.

Introduction to Pandas

Written by Luke Chang & Jin Cheong

Analyzing data requires being facile with manipulating and transforming datasets to be able to test specific hypotheses. Data come in all different types of flavors and there are many different tools in the Python ecosystem to work with pretty much any type of data you might encounter. For example, you might be interested in working with functional neuroimaging data that is four dimensional. Three dimensional matrices contain brain activations in space, and these data can change over time in the 4th dimension. This type of data is well suited for numpy and specialized brain imaging packages such as nilearn. The majority of data, however, is typically in some version of a two-dimensional observations by features format as might be seen in an excel spreadsheet, a SQL table, or in a comma delimited format (i.e., csv).

In Python, the Pandas library is a powerful tool to work with this type of data. This is a very large library with a tremendous amount of functionality. In this tutorial, we will cover the basics of how to load and manipulate data and will focus on common to data munging tasks.

For those interested in diving deeper into Pandas, there are many online resources. There is the Pandas online documention, stackoverflow, and medium blogposts. I highly recommend Jake Vanderplas's terrific Python Data Science Handbook. In addition, here is a brief video by Tal Yarkoni providing a useful introduction to pandas.

After the tutorial you will have the chance to apply the methods to a new set of data.

Pandas Objects

Pandas has several objects that are commonly used (i.e., Series, DataFrame, Index). At it's core, Pandas Objects are enhanced numpy arrays where columns and rows can have special names and there are lots of methods to operate on the data. See Jake Vanderplas's tutorial for a more in depth overview.

Series

A pandas `Series` is a one-dimensional array of indexed data.

```
import pandas as pd

data = pd.Series([1, 2, 3, 4, 5])
data
```

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

The indices can be integers like in the example above. Alternatively, the indices can be labels.

```
data = pd.Series([1,2,3], index=['a', 'b', 'c'])
data
```

```
a    1
b    2
c    3
dtype: int64
```

Also, `Series` can be easily created from dictionaries

```
data = pd.Series({'A':5, 'B':3, 'C':1})
data
```

```
A    5
B    3
C    1
dtype: int64
```

DataFrame

If a `Series` is a one-dimensional indexed array, the `DataFrame` is a two-dimensional indexed array. It can be thought of as a collection of Series objects, where each Series represents a column, or as an enhanced 2D numpy array.

In a `DataFrame`, the index refers to labels for each row, while columns describe each column.

First, let's create a `DataFrame` using random numbers generated from numpy.

```
import numpy as np

data = pd.DataFrame(np.random.random((5, 3)))
data
```

	0	1	2
0	0.126463	0.901287	0.487618
1	0.033185	0.287304	0.773930
2	0.069269	0.793656	0.025462
3	0.228061	0.559880	0.473307
4	0.261735	0.353904	0.880351

We could also initialize with column names

```
data = pd.DataFrame(np.random.random((5, 3)), columns=['A', 'B', 'C'])
data
```

	A	B	C
0	0.720483	0.350045	0.538831
1	0.877044	0.308761	0.786224
2	0.399717	0.683241	0.878268
3	0.565219	0.719122	0.410798
4	0.165874	0.667780	0.820636

Alternatively, we could create a `DataFrame` from multiple `Series` objects.

```
a = pd.Series([1, 2, 3, 4])
b = pd.Series(['a', 'b', 'c', 'd'])
data = pd.DataFrame({'Numbers':a, 'Letters':b})
data
```

	Numbers	Letters
0	1	a
1	2	b
2	3	c
3	4	d

Or a python dictionary

```
data = pd.DataFrame({'State':['California', 'Colorado', 'New Hampshire'],
                     'Capital':['Sacremento', 'Denver', 'Concord']})
data
```

	State	Capital
0	California	Sacremento
1	Colorado	Denver
2	New Hampshire	Concord

Loading Data

Loading data is fairly straightfoward in Pandas. Type `pd.read` then press tab to see a list of functions that can load specific file formats such as: csv, excel, spss, and sql.

In this example, we will use `pd.read_csv` to load a .csv file into a dataframe. Note that read_csv() has many options that can be used to make sure you load the data correctly. You can explore the docstrings for a function to get more information about the inputs and general useage guidelines by running `pd.read_csv?`

```
pd.read_csv?
```

To load a csv file we will need to specify either the relative or absolute path to the file.

The command `pwd` will print the path of the current working directory.

```
pwd
```

```
'/Users/lukechang/Dropbox/Dartbrains/Notebooks'
```

We will now load the Pandas has many ways to read data different data formats into a dataframe. Here we will use the `pd.read_csv` function.

```
df = pd.read_csv('/Users/lukechang/Dropbox/Dartbrains/Data/salary/salary.csv', sep = ',')
# df = pd.read_csv('psych60/data/salary/salary.csv', sep = ',')
```

Ways to check the dataframe

There are many ways to examine your dataframe. One easy way is to just call the dataframe variable itself.

```
df
```

	salary	gender	departm	years	age	publications
0	86285	0	bio	26.0	64.0	72
1	77125	0	bio	28.0	58.0	43
2	71922	0	bio	10.0	38.0	23
3	70499	0	bio	16.0	46.0	64
4	66624	0	bio	11.0	41.0	23
...	...	...	...	...	...	...
72	53662	1	neuro	1.0	31.0	3
73	57185	1	stat	9.0	39.0	7
74	52254	1	stat	2.0	32.0	9
75	61885	1	math	23.0	60.0	9
76	49542	1	math	3.0	33.0	5

77 rows × 6 columns

However, often the dataframes can be large and we may be only interested in seeing the first few rows. `df.head()` is useful for this purpose.

```
df.head()
```

	salary	gender	departm	years	age	publications
0	86285	0	bio	26.0	64.0	72
1	77125	0	bio	28.0	58.0	43
2	71922	0	bio	10.0	38.0	23
3	70499	0	bio	16.0	46.0	64
4	66624	0	bio	11.0	41.0	23

On the top row, you have column names, that can be called like a dictionary (a dataframe can be essentially thought of as a dictionary with column names as the keys). The left most column (0,1,2,3,4...) is called the index of the dataframe. The default index is sequential integers, but it can be set to anything as long as each row is unique (e.g., subject IDs)

```
print("Indexes")
print(df.index)
print("Columns")
print(df.columns)
print("Columns are like keys of a dictionary")
print(df.keys())
```

```
Indexes
RangeIndex(start=0, stop=77, step=1)
Columns
Index(['salary', 'gender', 'departm', 'years', 'age', 'publications'],
dtype='object')
Columns are like keys of a dictionary
Index(['salary', 'gender', 'departm', 'years', 'age', 'publications'],
dtype='object')
```

You can access the values of a column by calling it directly. Single bracket returns a `Series` and double bracket returns a `dataframe`.

Let's return the first 10 rows of salary.

```
df['salary'][:10]
```

```
0    86285
1    77125
2    71922
3    70499
4    66624
5    64451
6    64366
7    59344
8    58560
9    58294
Name: salary, dtype: int64
```

`shape` is another useful method for getting the dimensions of the matrix.

We will print the number of rows and columns in this data set using fstring formatting. First, you need to specify a string starting with 'f', like this `f'anything'`. It is easy to insert variables with curly brackets like this `f'rows: {rows}'`.

Here is more info about formatting text.

```
rows, cols = df.shape
print(f'There are {rows} rows and {cols} columns in this data set')
```

```
There are 77 rows and 6 columns in this data set
```

Describing the data

We can use the `.describe()` method to get a quick summary of the continuous values of the data frame. We will `.transpose()` the output to make it slightly easier to read.

```
df.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	
salary	77.0	67748.519481	15100.581435	44687.0	57185.0	62607.0	75382.0	11
gender	77.0	0.142857	0.387783	0.0	0.0	0.0	0.0	
years	76.0	14.973684	8.617770	1.0	8.0	14.0	23.0	
age	76.0	45.486842	9.005914	31.0	38.0	44.0	53.0	
publications	77.0	21.831169	15.240530	3.0	9.0	19.0	33.0	

We can also get quick summary of a pandas series, or specific column of a pandas dataframe.

```
df.departm.describe()
```

```
count      77
unique      7
top       bio
freq       16
Name: departm, dtype: object
```

Sometimes, you will want to know how many data points are associated with a specific variable for categorical data. The `value_counts` method can be used for this goal.

For example, how many males and females are in this dataset?

```
df['gender'].value_counts()
```

```
0    67
1     9
2     1
Name: gender, dtype: int64
```

You can see that there are more than 2 genders specified in our data.

This is likely an error in the data collection process. It's always up to the data analyst to decide what to do in these cases. Because we don't know what the true value should have been, let's just remove the row from the dataframe by finding all rows that are not '2'.

```
df = df.loc[df['gender']!=2]

df['gender'].value_counts()
```

```
0    67
1     9
Name: gender, dtype: int64
```

Dealing with missing values

Data are always messy and often have lots of missing values. There are many different ways, in which missing data might present `NaN`, `None`, or `NA`, Sometimes researchers code missing values with specific numeric codes such as 999999. It is important to find these as they can screw up your analyses if they are hiding in your data.

If the missing values are using a standard pandas or numpy value such as `NaN`, `None`, or `NA`, we can identify where the missing values are as booleans using the `isnull()` method.

The `isnull()` method will return a dataframe with True/False values on whether a datapoint is null or not a number (nan).

```
df.isnull()
```

	salary	gender	departm	years	age	publications
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	False	False
...	...	...	...	...	...	...
72	False	False	False	False	False	False
73	False	False	False	False	False	False
74	False	False	False	False	False	False
75	False	False	False	False	False	False
76	False	False	False	False	False	False

76 rows × 6 columns

Suppose we wanted to count the number of missing values for each column in the dataset.

One thing that is nice about Python is that you can chain commands, which means that the output of one method can be the input into the next method. This allows us to write intuitive and concise code. Notice how we take the `sum()` of all of the null cases. We can chain the `.null()` and `.sum()` methods to see how many null values are added up in each column.

```
df.isnull().sum()
```

```
salary          0
gender          0
departm         0
years           1
age             1
publications    0
dtype: int64
```

You can use the boolean indexing once again to see the datapoints that have missing values. We chained the method `.any()` which will check if there are any True values for a given axis. Axis=0 indicates rows, while Axis=1 indicates columns. So here we are creating a boolean index for row where *any* column has a missing value.

```
df[df.isnull().any(axis=1)]
```

	salary	gender	departm	years	age	publications
18	64762	0	chem	25.0	NaN	29
24	104828	0	geol	NaN	50.0	44

You may look at where the values are not null. Note that indexes 18, and 24 are missing.

```
df[~df.isnull().any(axis=1)]
```

	salary	gender	departm	years	age	publications
0	86285	0	bio	26.0	64.0	72
1	77125	0	bio	28.0	58.0	43
2	71922	0	bio	10.0	38.0	23
3	70499	0	bio	16.0	46.0	64
4	66624	0	bio	11.0	41.0	23
...	...	...	...	...	...	...
72	53662	1	neuro	1.0	31.0	3
73	57185	1	stat	9.0	39.0	7
74	52254	1	stat	2.0	32.0	9
75	61885	1	math	23.0	60.0	9
76	49542	1	math	3.0	33.0	5

74 rows × 6 columns

There are different techniques for dealing with missing data. An easy one is to simply remove rows that have any missing values using the `dropna()` method.

```
df.dropna(inplace=True)
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

Now we can check to make sure the missing rows are removed. Let's also check the new dimensions of the dataframe.

```
rows, cols = df.shape
print(f'There are {rows} rows and {cols} columns in this data set')

df.isnull().sum()
```

```
There are 74 rows and 6 columns in this data set
```

```
salary          0
gender          0
departm         0
years           0
age             0
publications    0
dtype: int64
```

Create New Columns

You can create new columns to fit your needs. For instance you can set initialize a new column with zeros.

```
df['pubperyear'] = 0
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

Here we can create a new column pubperyear, which is the ratio of the number of papers published per year

```
df['pubperyear'] = df['publications']/df['years']
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

Indexing and slicing Data

Indexing in Pandas can be tricky. There are many ways to index in pandas, for this tutorial we will focus on four: loc, iloc, boolean, and indexing numpy values. For a more in depth overview see Jake Vanderplas's tutorial] (https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/03.02-Data-Indexing-and-Selection.ipynb), where he also covers more advanced topics, such as hierarchical indexing.

Indexing with Keys

First, we will cover indexing with keys using the `.loc` method. This method references the explicit index with a key name. It works for both index names and also column names. Note that often the keys for rows are integers by default.

In this example, we will return rows 10-20 on the salary column.

```
df.loc[10:20, 'salary']
```

```
10    56092
11    54452
12    54269
13    55125
14    97630
15    82444
16    76291
17    75382
19    62607
20    60373
Name: salary, dtype: int64
```

You can return multiple columns using a list.

```
df.loc[:10, ['salary', 'publications']]
```

	salary	publications
0	86285	72
1	77125	43
2	71922	23
3	70499	64
4	66624	23
5	64451	44
6	64366	22
7	59344	11
8	58560	8
9	58294	12
10	56092	4

Indexing with Integers

Next we wil try `.iloc`. This method references the implicit python index using integer indexing (starting from 0, exclusive of last number). You can think of this like row by column indexing using integers.

For example, let's grab the first 3 rows and columns.

```
df.iloc[0:3, 0:3]
```

	salary	gender	departm
0	86285	0	bio
1	77125	0	bio
2	71922	0	bio

Let's make a new data frame with just Males and another for just Females. Notice, how we added the `.reset_index(drop=True)` method? This is because assigning a new dataframe based on indexing another dataframe will retain the *original* index. We need to explicitly tell pandas to reset the index if we want it to start from zero.

```
male_df = df[df.gender == 0].reset_index(drop=True)
female_df = df[df.gender == 1].reset_index(drop=True)
```

Indexing with booleans

Boolean or logical indexing is useful if you need to sort the data based on some True or False value.

For instance, who are the people with salaries greater than 90K but lower than 100K ?

```
df[ (df.salary > 90000) & (df.salary < 100000)]
```

	salary	gender	departm	years	age	publications	pubperyear
14	97630	0	chem	34.0	64.0	43	1.264706
30	92951	0	neuro	11.0	41.0	20	1.818182
54	96936	0	physics	15.0	50.0	17	1.133333

This also works with the `.loc` method, which is what you need to do if you want to return specific columns

```
df.loc[ (df.salary > 90000) & (df.salary < 100000), ['salary', 'gender']]
```

	salary	gender
14	97630	0
30	92951	0
54	96936	0

Numpy indexing

Finally, you can also return a numpy matrix from a pandas data frame by accessing the `.values` property. This returns a numpy array that can be indexed using numpy integer indexing and slicing.

As an example, let's grab the last 10 rows and the first 3 columns.

```
df.values[-10:, :3]
```

```
array([[53638, 0, 'math'],
       [59139, 1, 'bio'],
       [52968, 1, 'bio'],
       [55949, 1, 'chem'],
       [58893, 1, 'neuro'],
       [53662, 1, 'neuro'],
       [57185, 1, 'stat'],
       [52254, 1, 'stat'],
       [61885, 1, 'math'],
       [49542, 1, 'math']], dtype=object)
```

Renaming

Part of cleaning up the data is renaming with more sensible names. This is easy to do with Pandas.

Renaming Columns

We can rename columns with the `.rename` method by passing in a dictionary using the `{'Old Name':'New Name'}`. We either need to assigne the result to a new variable or add `inplace=True`.

```
df.rename({'departm':'department','pubperyear':'pub_per_year'}, axis=1, inplace=True)
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py:4133:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  errors=errors,
```

Renaming Rows

Often we may want to change the coding scheme for a variable. For example, it is hard to remember what zeros and ones mean in the gender variable. We can make this easier by changing these with a dictionary `{0:'male', 1:'female'}` with the `replace` method. We can do this `inplace=True` or we can assign it to a new variable. As an example, we will assign this to a new variable to also retain the original lablels.

```
df['gender_name'] = df['gender'].replace({0:'male', 1:'female'})

df.head()
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

	salary	gender	department	years	age	publications	pub_per_year	gender_name
0	86285	0	bio	26.0	64.0	72	2.769231	male
1	77125	0	bio	28.0	58.0	43	1.535714	male
2	71922	0	bio	10.0	38.0	23	2.300000	male
3	70499	0	bio	16.0	46.0	64	4.000000	male
4	66624	0	bio	11.0	41.0	23	2.090909	male

Operations

One of the really fun things about pandas once you get the hang of it is how easy it is to perform operations on the data. It is trivial to compute simple summaries of the data. We can also leverage the object-oriented nature of a pandas object, we can chain together multiple commands.

For example, let's grab the mean of a few columns.

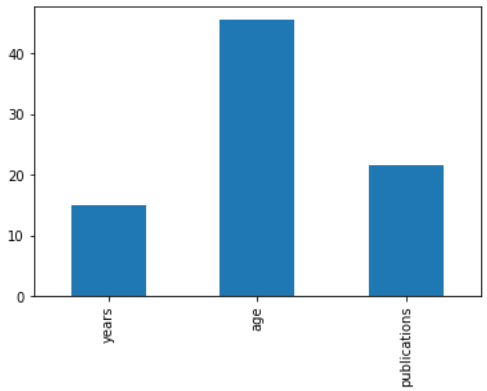
```
df.loc[:,['years', 'age', 'publications']].mean()
```

```
years           14.972973
age             45.567568
publications    21.662162
dtype: float64
```

We can also turn these values into a plot with the `plot` method, which we will cover in more detail in future tutorials.

```
%matplotlib inline

df.loc[:,['years', 'age', 'publications']].mean().plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fea789a8810>
```



Perhaps we want to see if there are any correlations in our dataset. We can do this with the `.corr` method.

```
df.corr()
```

	salary	gender	years	age	publications	pub_per_year
salary	1.000000	-0.300071	0.303275	0.275534	0.427426	-0.016988
gender	-0.300071	1.000000	-0.275468	-0.277098	-0.249410	0.024210
years	0.303275	-0.275468	1.000000	0.958181	0.323965	-0.541125
age	0.275534	-0.277098	0.958181	1.000000	0.328285	-0.498825
publications	0.427426	-0.249410	0.323965	0.328285	1.000000	0.399865
pub_per_year	-0.016988	0.024210	-0.541125	-0.498825	0.399865	1.000000

Merging Data

Another common data manipulation goal is to merge datasets. There are multiple ways to do this in pandas, we will cover concatenation, append, and merge.

Concatenation

Concatenation describes the process of *stacking* dataframes together. The main thing to consider is to make sure that the shapes of the two dataframes are the same as well as the index labels. For example, if we wanted to vertically stack two dataframe, they need to have the same column names.

Remember that we previously created two separate dataframes for males and females? Let's put them back together using the `pd.concat` method. Note how the index of this output retains the old index.

```
combined_data = pd.concat([female_df, male_df], axis = 0)
```

We can reset the index using the `reset_index` method.

```
pd.concat([male_df, female_df], axis = 0).reset_index(drop=True)
```

	salary	gender	departm	years	age	publications	pubperyear
0	86285	0	bio	26.0	64.0	72	2.769231
1	77125	0	bio	28.0	58.0	43	1.535714
2	71922	0	bio	10.0	38.0	23	2.300000
3	70499	0	bio	16.0	46.0	64	4.000000
4	66624	0	bio	11.0	41.0	23	2.090909
...	...	...	...	...	...	...	...
69	53662	1	neuro	1.0	31.0	3	3.000000
70	57185	1	stat	9.0	39.0	7	0.777778
71	52254	1	stat	2.0	32.0	9	4.500000
72	61885	1	math	23.0	60.0	9	0.391304
73	49542	1	math	3.0	33.0	5	1.666667

74 rows × 7 columns

We can also concatenate columns in addition to rows. Make sure that the number of rows are the same in each dataframe. For this example, we will just create two new data frames with a subset of the columns and then combine them again.

```
df1 = df[['salary', 'gender']]
df2 = df[['age', 'publications']]
df3 = pd.concat([df1, df2], axis=1)
df3.head()
```

	salary	gender	age	publications
0	86285	0	64.0	72
1	77125	0	58.0	43
2	71922	0	38.0	23
3	70499	0	46.0	64
4	66624	0	41.0	23

Append

We can also combine datasets by appending new data to the end of a dataframe.

Suppose we want to append a new data entry of an additional participant onto the `df3` dataframe. Notice that we need to specify to `ignore_index=True` and also that we need to assign the new dataframe back to a variable. This operation is not done in place.

For more information about concatenation and appending see Jake Vanderplas's tutorial.

```
new_data = pd.Series({'salary':100000, 'gender':1, 'age':38, 'publications':46})
df3 = df3.append(new_data, ignore_index=True)
df3.tail()
```

	salary	gender	age	publications
70	57185	1	39.0	7
71	52254	1	32.0	9
72	61885	1	60.0	9
73	49542	1	33.0	5
74	100000	1	38.0	46

Merge

The most powerful method of merging data is using the `pd.merge` method. This allows you to merge datasets of different shapes and sizes on specific variables that match. This is very common when you need to merge multiple sql tables together for example.

In this example, we are creating two separate data frames that have different states and columns and will merge on the `State` column.

First, we will only retain rows where there is a match across dataframes, using `how=inner`. This is equivalent to an 'and' join in sql.

```
df1 = pd.DataFrame({'State':['California', 'Colorado', 'New Hampshire'],
                    'Capital':['Sacremento', 'Denver', 'Concord']})
df2 = pd.DataFrame({'State':['California', 'New Hampshire', 'New York'],
                    'Population':['39512223', '1359711', '19453561']})

df3 = pd.merge(left=df1, right=df2, on='State', how='inner')
df3
```

	State	Capital	Population
0	California	Sacremento	39512223
1	New Hampshire	Concord	1359711

Notice how there are only two rows in the merged dataframe.

We can also be more inclusive and match on `State` column, but retain all rows. This is equivalent to an 'or' join.

```
df3 = pd.merge(left=df1, right=df2, on='State', how='outer')
df3
```

	State	Capital	Population
0	California	Sacremento	39512223
1	Colorado	Denver	NaN
2	New Hampshire	Concord	1359711
3	New York	NaN	19453561

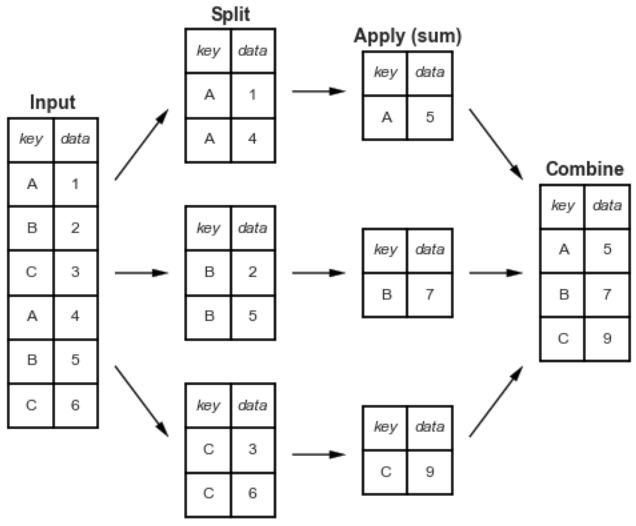
This is a very handy way to merge data when you have lots of files with missing data. See Jake Vanderplas's tutorial for a more in depth overview.

Grouping

We've seen above that it is very easy to summarize data over columns using the builtin functions such as `pd.mean()`. Sometimes we are interested in summarizing data over different groups of rows. For example, what is the mean of participants in Condition A compared to Condition B?

This is suprisingly easy to compute in pandas using the `groupby` operator, where we aggregate data using a specific operation over different labels.

One useful way to conceptualize this is using the **Split, Apply, Combine** operation (similar to map-reduce).



This figure is taken from Jake Vanderplas's tutorial and highlights how input data can be *split* on some key and then an operation such as sum can be *applied* separately to each split. Finally, the results of the applied function for each key can be *combined* into a new data frame.

Groupby

In this example, we will use the `groupby` operator to split the data based on gender labels and separately calculate the mean for each group.

```
df.groupby('gender_name').mean()
```

gender_name	salary	gender	years	age	publications	pub_per_year
female	55719.666667	1.0	8.666667	38.888889	11.555556	2.043170
male	69108.492308	0.0	15.846154	46.492308	23.061538	1.924709

Other default aggregation methods include `.count()`, `.mean()`, `.median()`, `.min()`, `.max()`, `.std()`, `.var()`, and `.sum()`

Transform

While the split, apply, combine operation that we just demonstrated is extremely usefuly to quickly summarize data based on a grouping key, the resulting data frame is compressed to one row per grouping label.

Sometimes, we would like to perform an operation over groups, but retain the original data shape. One common example is standardizing data within a subject or grouping variable. Normally, you might think to loop over subject ids and separately z-score or center a variable and then recombine the subject data using a vertical concatenation operation.

The `transform` method in pandas can make this much easier and faster!

Suppose we want to compute the standardized salary separately for each department. We can standardize using a z-score which requires subtracting the departmental mean from each professor's salary in that department, and then dividing it by the departmental standard deviation.

We can do this by using the `groupby(key)` method chained with the `.transform(function)` method. It will group the dataframe by the key column, perform the "function" transformation of the data and return data in same format. We can then assign the results to a new column in the dataframe.

```
df['salary_dept_z'] = (df['salary'] - df.groupby('department').transform('mean')
['salary'])/df.groupby('department').transform('std')['salary']
df.head()
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

	salary	gender	department	years	age	publications	pub_per_year	gender_name	salary_
0	86285	0	bio	26.0	64.0	72	2.769231	male	2.4
1	77125	0	bio	28.0	58.0	43	1.535714	male	1.4
2	71922	0	bio	10.0	38.0	23	2.300000	male	0.9
3	70499	0	bio	16.0	46.0	64	4.000000	male	0.7
4	66624	0	bio	11.0	41.0	23	2.090909	male	0.3

This worked well, but is also pretty verbose. We can simplify the syntax a little bit more using a `lambda` function, where we can define the zscore function.

```
calc_zscore = lambda x: (x - x.mean()) / x.std()

df['salary_dept_z'] = df['salary'].groupby(df['department']).transform(calc_zscore)

df.head()
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  This is separate from the ipykernel package so we can avoid doing imports until
```

	salary	gender	department	years	age	publications	pub_per_year	gender_name	salary_
0	86285	0	bio	26.0	64.0	72	2.769231	male	2.4
1	77125	0	bio	28.0	58.0	43	1.535714	male	1.4
2	71922	0	bio	10.0	38.0	23	2.300000	male	0.9
3	70499	0	bio	16.0	46.0	64	4.000000	male	0.7
4	66624	0	bio	11.0	41.0	23	2.090909	male	0.3

For a more in depth overview of data aggregation and grouping, check out Jake Vanderplas's tutorial

Reshaping Data

The last topic we will cover in this tutorial is reshaping data. Data is often in the form of observations by features, in which there is a single row for each independent observation of data and a separate column for each feature of the data. This is commonly referred to as as the *wide* format. However, when running regression or plotting in libraries such as seaborn, we often want our data in the *long* format, in which each grouping variable is specified in a separate column.

In this section we cover how to go from wide to long using the `melt` operation and from long to wide using the `pivot` function.

Melt

To `melt` a dataframe into the long format, we need to specify which variables are the `id_vars`, which ones should be combined into a `value_var`, and finally, what we should label the column name for the `value_var`, and also for the `var_name`. We will call the values 'Ratings' and variables 'Condition'.

First, we need to create a dataset to play with.

```
data = pd.DataFrame(np.vstack([np.arange(1,6), np.random.random((3, 5))]).T, columns=
['ID','A', 'B', 'C'])
data
```

	ID	A	B	C
0	1.0	0.330666	0.893955	0.524217
1	2.0	0.881766	0.722020	0.895915
2	3.0	0.804144	0.316754	0.357837
3	4.0	0.830743	0.550464	0.704257
4	5.0	0.881331	0.755053	0.811475

Now, let's melt the dataframe into the long format.

```
df_long = pd.melt(data,id_vars='ID', value_vars=['A', 'B', 'C'], var_name='Condition',
value_name='Rating')
df_long
```

	ID	Condition	Rating
0	1.0	A	0.330666
1	2.0	A	0.881766
2	3.0	A	0.804144
3	4.0	A	0.830743
4	5.0	A	0.881331
5	1.0	B	0.893955
6	2.0	B	0.722020
7	3.0	B	0.316754
8	4.0	B	0.550464
9	5.0	B	0.755053
10	1.0	C	0.524217
11	2.0	C	0.895915
12	3.0	C	0.357837
13	4.0	C	0.704257
14	5.0	C	0.811475

Notice how the id variable is repeated for each condition?

Pivot

We can also go back to the wide data format from a long dataframe using `pivot`. We just need to specify the variable containing the labels which will become the `columns` and the `values` column that will be broken into separate columns.

```
df_wide = df_long.pivot(index='ID', columns='Condition', values='Rating')
df_wide
```

Condition	A	B	C
ID			
1.0	0.330666	0.893955	0.524217
2.0	0.881766	0.722020	0.895915
3.0	0.804144	0.316754	0.357837
4.0	0.830743	0.550464	0.704257
5.0	0.881331	0.755053	0.811475

Exercises (Homework)

The following exercises uses the dataset “salary_exercise.csv” adapted from material available here

These are the salary data used in Weisberg’s book, consisting of observations on six variables for 52 tenure-track professors in a small college. The variables are:

- sx = Sex, coded 1 for female and 0 for male
- rk = Rank, coded
- 1 for assistant professor,
- 2 for associate professor, and
- 3 for full professor
- yr = Number of years in current rank
- dg = Highest degree, coded 1 if doctorate, 0 if masters
- yd = Number of years since highest degree was earned
- sl = Academic year salary, in dollars.

Reference: S. Weisberg (1985). Applied Linear Regression, Second Edition. New York: John Wiley and Sons. Page 194.

Exercise 1

Read the salary_exercise.csv into a dataframe, and change the column names to a more readable format such as sex, rank, yearsinrank, degree, yearssinceHD, and salary.

Clean the data by excluding rows with any missing value.

What are the overall mean, standard deviation, min, and maximum of professors' salary?

Exercise 2

Create two separate dataframes based on the type of degree. Now calculate the mean salary of the 5 oldest professors of each degree type.

Exercise 3

What is the correlation between the standardized salary *across* all ranks and the standardized salary *within* ranks?

Introduction to Plotting

Written by Luke Chang & Jin Cheong

In this lab, we will introduce the basics of plotting in python using the `matplotlib` and `seaborn` packages. Matlotlib is probably the most popular python package for 2D graphics and has a nice tradeoff between ease of use and customizability. We will be working with the `pyplot` interface, which is an object-oriented plotting library based on plotting in Matlab. Many graphics libraries are built on top of matplotlib, and have tried to make plotting even easier. One library that is very nice to generate plots similar to how analyses are performed is seaborn. There are many great tutorials online. Here are a few that I found to be helpful from neurohackademy, Jake Vanderplas, and rougier.

Matplotlib Key Concepts

There are a few different types of concepts in the `matplotlib.pyplot` framework.

- `Figure`: Essentially the canvas, which contains the plots
- `Axes`: An individual plot within a figure. Each Axes has a title, an x-label, and a y-label
- `Axis`: These contain the graph limits and tickmarks
- `Artist`: Everything that is visiable on the figure is an artist (e.g., Text objects, Line2D object, collection objects). Artists are typically tied to a specific Axes.

Note: `%matplotlib inline` is an example of 'cell magic' and enables plotting *within* the notebook and not opening a separate window. In addition, you may want to try using `%matplotlib notebook`, which will allow more interactive plotting.

Let's get started by loading the modules we will use for this tutorial.

```
%matplotlib inline

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Lineplot

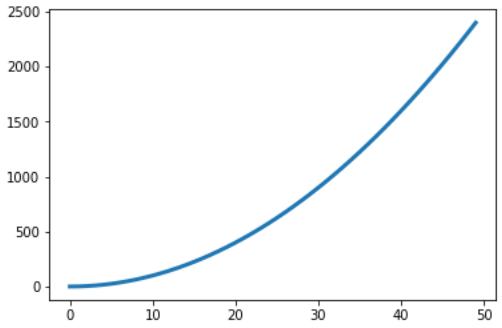
First, let's generate some numbers and create a basic lineplot.

In this example, we plot:

$y = x^2$

```
x = np.arange(0, 50, 1)
y = x ** 2

plt.plot(x, y, linewidth=3)
```

```
[<matplotlib.lines.Line2D at 0x7f7f1053e450>]
```

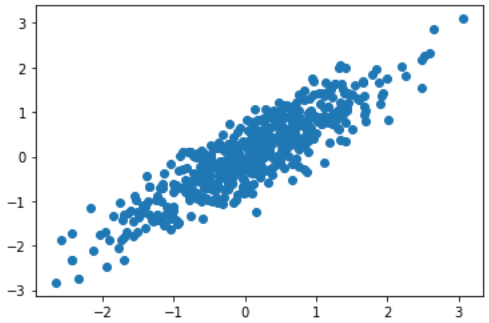


Scatterplot

We can also plot associations between two different variables using a scatterplot. In this example, we will simulate correlated data with $\mathcal{N}(0, 1)$ using `np.random.multivariate_normal` and create a scatterplot. Try playing with the covariance values to change the degree of association.

```
n = 500
r = .9
mu = np.array([0, 0])
cov = np.array([
    [1, r],
    [r, 1]])
data = np.random.multivariate_normal(mu, cov, size=n)

plt.scatter(data[:,0], data[:,1])
```

```
<matplotlib.collections.PathCollection at 0x7f7f4b304550>
```



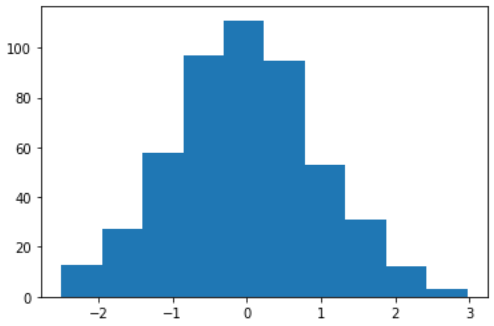
Histogram

We can also plot the distribution of a variable using a histogram.

Let's see if the data we simulated above are actually normally distributed.

```
plt.hist(data[:,0])
```

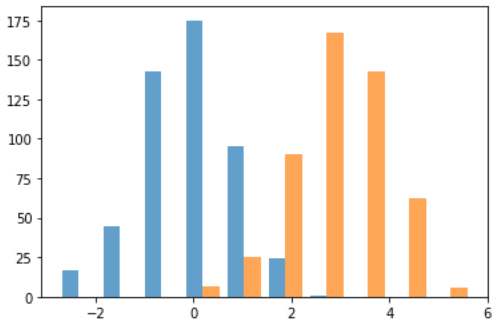
```
(array([ 13.,  27.,  58.,  97., 111.,  95.,  53.,  31.,  12.,   3.]),
 array([-2.49785684, -1.95098132, -1.40410579, -0.85723027, -0.31035474,
         0.23652078,  0.78339631,  1.33027183,  1.87714736,  2.42402288,
         2.9708984 ]),
 <a list of 10 Patch objects>)
```



We can also plot overlaying histograms. Let's simulate more data, but shift the mean of one by 3.

```
r = .9
mu = np.array([0, 3])
cov = np.array([
    [1, r],
    [r, 1]])
data = np.random.multivariate_normal(mu, cov, size=n)

plt.hist(data, alpha=.7)
```

```
(array([[ 17.,  45., 143., 175.,  95.,  24.,   1.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   7.,  25.,  90., 167., 143.,  62.,   6.]]),
 array([-2.77829094, -1.93245112, -1.0866113 , -0.24077149,  0.60506833,
         1.45090815,  2.29674797,  3.14258779,  3.9884276 ,  4.83426742,
         5.68010724]),
 <a list of 2 Lists of Patches objects>)
```

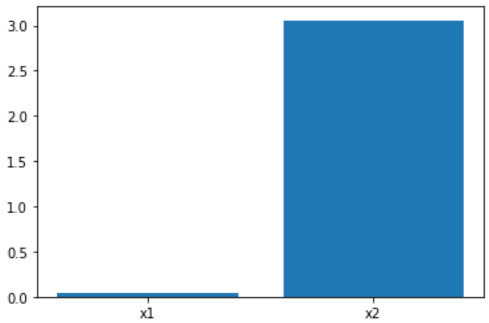


Bar Plot

We can also plot the same data as a bar plot to emphasize the difference in the means of the distributions. To create a bar plot, we need to specify the bar names and the heights of the bars.

```
plt.bar(['x1','x2'], np.mean(data, axis=0))
```

```
<BarContainer object of 2 artists>
```

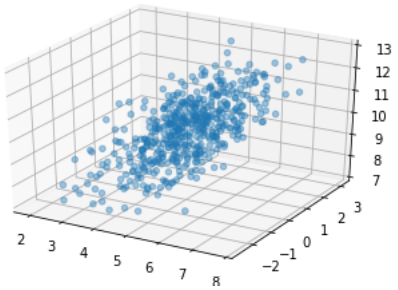


3D Plots

We can also plot in 3 dimensions with `mplot3d`. Here we will simulate 3 different variables with different correlations.

```
from mpl_toolkits import mplot3d

n = 500

r1 = .1
r2 = .5
r3 = .9

mu = np.array([5.0, 0.0, 10.0])

cov = np.array([
        [  1, r1, r2],
        [ r1,  1,  r3],
        [ r2,  r3,  1]
    ])
data = np.random.multivariate_normal(mu, cov, size=n)


ax = plt.axes(projection='3d')
ax.scatter3D(data[:,0], data[:,1], data[:,2], alpha=.4)
```

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f7f087ed750>
```



Customization

One of the nice things about matplotlib is that everything is customizable.

Let's go back to our first scatterplot and show how we can customize it to make it easier to read.

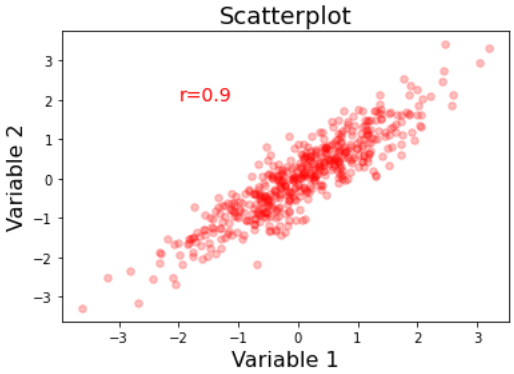
We can specify the type of `marker`, the `color`, the `alpha` transparency, and the size of the dots with `s`.

We can also label the axes with `xlabel` and `ylabel`, and add a `title`.

Finally, we can add text annotations, such as the strength of the correlation with `annotate`.

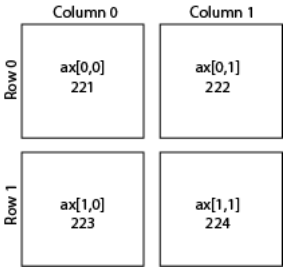
```
n = 500
r = .9
mu = np.array([0, 0])
cov = np.array([
    [1, r],
    [r, 1]])
data = np.random.multivariate_normal(mu, cov, size=n)

plt.scatter(data[:,0], data[:,1], color='r', marker='o', s=30, alpha=.25)

plt.xlabel('Variable 1', fontsize=16)
plt.ylabel('Variable 2', fontsize=16)
plt.title('Scatterplot', fontsize=18)
plt.annotate(f'r={r}', xy=(-2,2), xycoords='data', fontsize=14, color='red')
```

```
Text(-2, 2, 'r=0.9')
```



Layouts

The easiest way to make customized layouts that can include multiple panels of a plot are with `subplot`.



There are two different ways to index. One is by adding a subplot. The first number is the number of rows, the second is the number of columns, and the third is the index number.

I personally prefer to index directly into the `ax` object with rows and columns as I find it more intuitive.

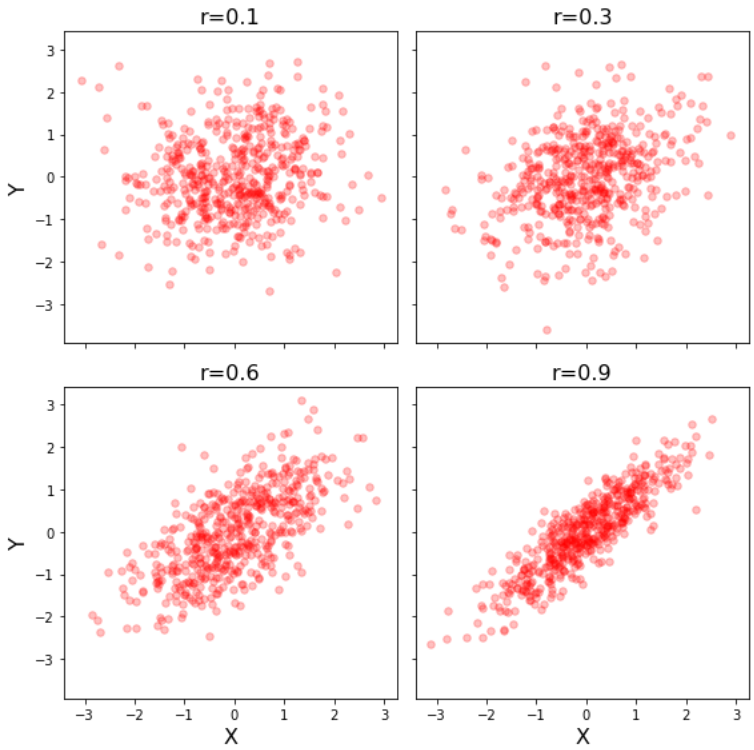
You can do even more advanced layouts with panels of different sizes using gridspec.

Let's make our simulation code into a function and use subplots to plot multiple panels.

We specify the number of rows and columns when we initialize the plot. We can also play with the size of the plot. Here we tell matplotlib that the x and y axes will be shared across the different panels. Finally, `plt.tight_layout()` helps keep everything formatted and organized nicely.

When modifying `axes` we need to use the `set_{}` command rather than just the command itself. For example, `ax[0,0].set_xlabel('X')` rather than `plt.xlabel('X')`.

```
def simulate_xy(n=500, r=.9):
    mu = np.array([0, 0])
    cov = np.array([
        [1, r],
        [r, 1]])
    return np.random.multivariate_normal(mu, cov, size=n)

f, ax = plt.subplots(nrows=2, ncols=2, figsize=(8, 8), sharex=True, sharey=True)

r=.1
sim1 = simulate_xy(n=500, r=r)
ax[0,0].scatter(sim1[:,0], sim1[:,1], color='r', marker='o', s=30, alpha=.25)
ax[0,0].set_title(f'r={r}', fontsize=16)
ax[0,0].set_ylabel('Y', fontsize=16)

r=.3
sim1 = simulate_xy(n=500, r=r)
ax[0,1].scatter(sim1[:,0], sim1[:,1], color='r', marker='o', s=30, alpha=.25)
ax[0,1].set_title(f'r={r}', fontsize=16)

r=.6
sim1 = simulate_xy(n=500, r=r)
ax[1,0].scatter(sim1[:,0], sim1[:,1], color='r', marker='o', s=30, alpha=.25)
ax[1,0].set_title(f'r={r}', fontsize=16)
ax[1,0].set_xlabel('X', fontsize=16)
ax[1,0].set_ylabel('Y', fontsize=16)

r=.9
sim1 = simulate_xy(n=500, r=r)
ax[1,1].scatter(sim1[:,0], sim1[:,1], color='r', marker='o', s=30, alpha=.25)
ax[1,1].set_title(f'r={r}', fontsize=16)
ax[1,1].set_xlabel('X', fontsize=16)

plt.tight_layout()
```



Saving Plots

Plots can be saved to disk using the `savefig` command. There are lots of ways to customize the saved plot. I typically save rasterized versions as `.png` and vectorized versions as `.pdf`. Don't forget to specify a path where you want the file written to.

```
plt.savefig('MyFirstPlot.png')

plt.savefig('/Users/lukechang/Downloads/MyFirstPlot.pdf')
```

```
<Figure size 432x288 with 0 Axes>
```

Seaborn

Seaborn is a plotting library built on Matplotlib that has many pre-configured plots that are often used for visualization. Other great tutorials about seaborn are here

Most seaborn plots can be customized using the standard matplotlib commands, though be sure to look at the docstrings first as often there are already keywords within each type of plot to do what you want.

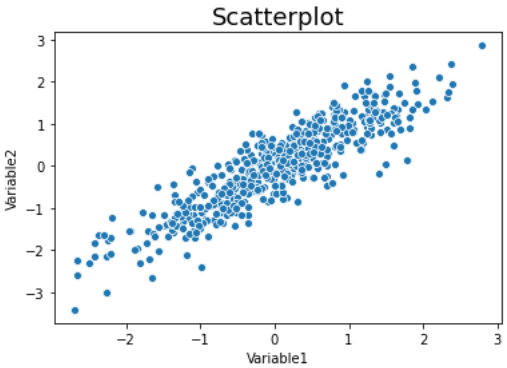
Scatterplots

There are many variants of each type of plot that you might like to use. For example, `scatterplot`, `regplot`, `jointplot`.

Let's update our simulation code a little bit to make this easier.

```
def simulate_xy(n=500, r=.9):
    mu = np.array([0, 0])
    cov = np.array([
        [1, r],
        [r, 1]])
    return pd.DataFrame(np.random.multivariate_normal(mu, cov, size=n), columns=
['Variable1', 'Variable2'])

sns.scatterplot(data=simulate_xy(), x='Variable1', y='Variable2')
plt.title('Scatterplot', fontsize=18)
```

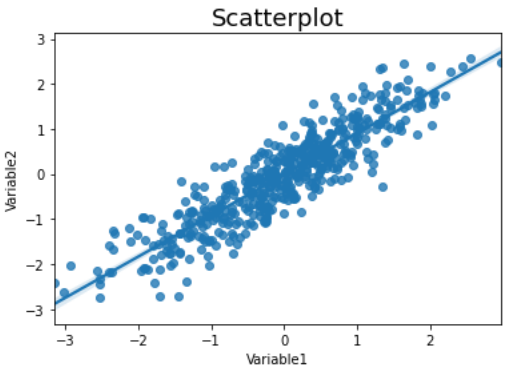
```
Text(0.5, 1.0, 'Scatterplot')
```



We can add regression lines with `regplot`.

```
sns.regplot(data=simulate_xy(), x='Variable1', y='Variable2')
plt.title('Scatterplot', fontsize=18)
```

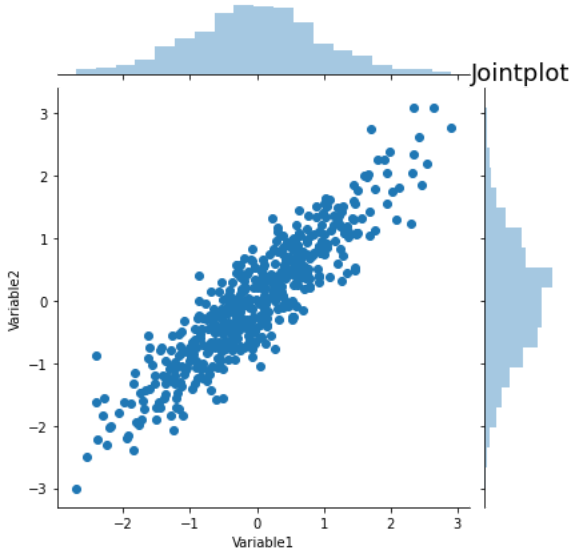
```
Text(0.5, 1.0, 'Scatterplot')
```



We can add histograms to show the distributions of x and y with `jointplot`.

```
sns.jointplot(data=simulate_xy(), x='Variable1', y='Variable2')
plt.title('Jointplot', fontsize=18)
```

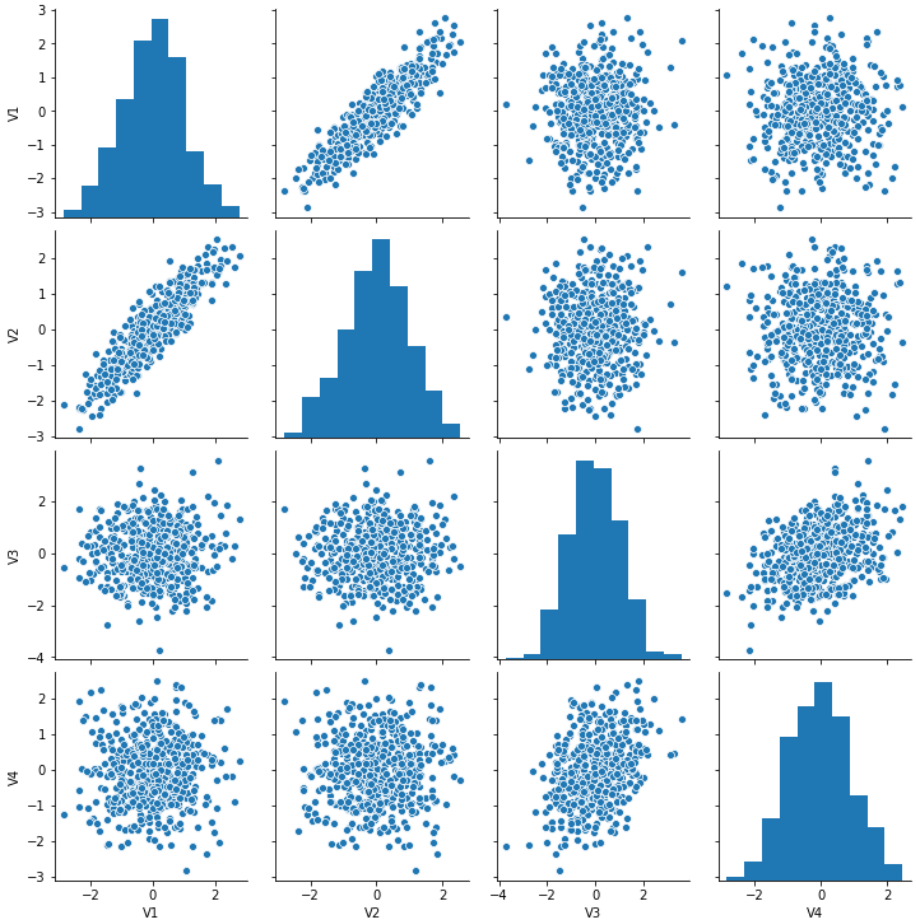
```
Text(0.5, 1.0, 'Jointplot')
```



We can create a quick way to view relations between multiple variables using `pairplot`.

```
data = pd.concat([simulate_xy(r=.9),simulate_xy(r=.3)], axis=1)
data.columns=['V1', 'V2', 'V3', 'V4']
sns.pairplot(data=data)
```

```
<seaborn.axisgrid.PairGrid at 0x7f7f28b75d10>
```



Factor plots

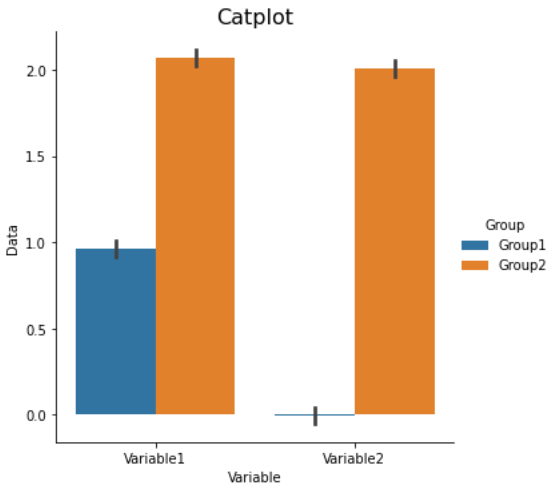
Factor plots allow you to visualize the distribution of parameters in different forms such as point, bar, or violin graphs.

One important thing to note is that the data needs to be in the long format. Let's quickly simulate some data to plot.

Here are some possible values for kind : {point, bar, count, box, violin, strip}

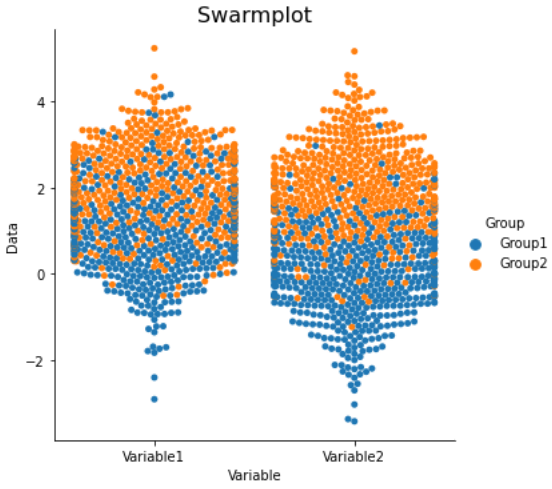
```
data1 = simulate_xy(r=.9)
data1['Group'] = 'Group1'
data1['Variable1'] = data1['Variable1'] + 1

data2 = simulate_xy(r=.3) + 2
data2['Group'] = 'Group2'

data = pd.concat([data1, data2], axis=0)
data_long = data.melt(id_vars='Group', value_vars=['Variable1', 'Variable2'],
var_name='Variable', value_name='Data')

sns.catplot(x='Variable', y='Data', hue='Group', data=data_long, ci=68, kind='bar')
plt.title('Catplot', fontsize=16)
```

```
Text(0.5, 1.0, 'Catplot')
```



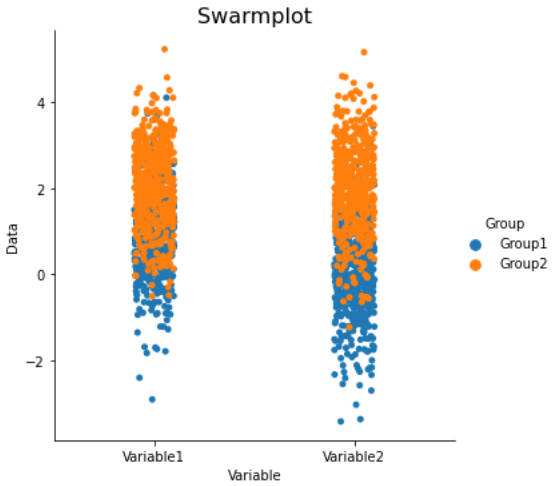

```
sns.catplot(x='Variable', y='Data', hue='Group', data=data_long, ci=68, kind='swarm')
plt.title('Swarmplot', fontsize=16)
```

```
Text(0.5, 1.0, 'Swarmplot')
```



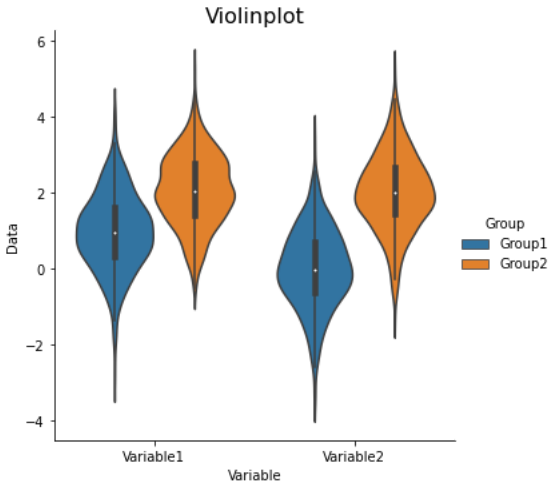

```
sns.catplot(x='Variable', y='Data', hue='Group', data=data_long, ci=68, kind='strip')
plt.title('Stripplot', fontsize=16)
```

```
Text(0.5, 1.0, 'Swarmplot')
```




```
sns.catplot(x='Variable', y='Data', hue='Group', data=data_long, ci=68, kind='violin')
plt.title('Violinplot', fontsize=16)
```

```
Text(0.5, 1.0, 'Violinplot')
```



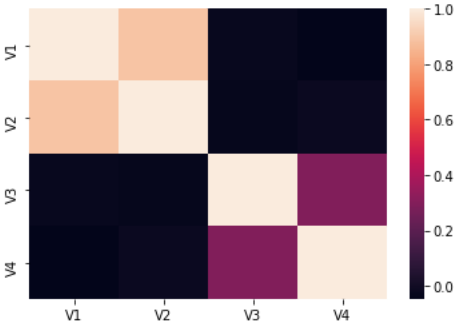
Heatmaps

Heatmap plots allow you to visualize matrices such as correlation matrices that show relationships across multiple variables.

Let's create a dataset with different relationships between variables. We can quickly calculate the correlation between these variables and visualize it with a heatmap

```
data = pd.concat([simulate_xy(r=.9), simulate_xy(r=.3)], axis=1)

data.columns=['V1', 'V2', 'V3', 'V4']
corr = data.corr()
sns.heatmap(data=corr)
```

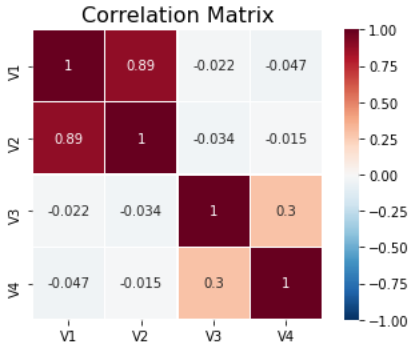
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7f09b933d0>
```



Like all other plots, we can also customize heatmaps.

```
sns.heatmap(corr, square=True, annot=True, linewidths=.5, cmap='RdBu_r', vmin=-1,
vmax=1)
plt.title('Correlation Matrix', fontsize=16)
```

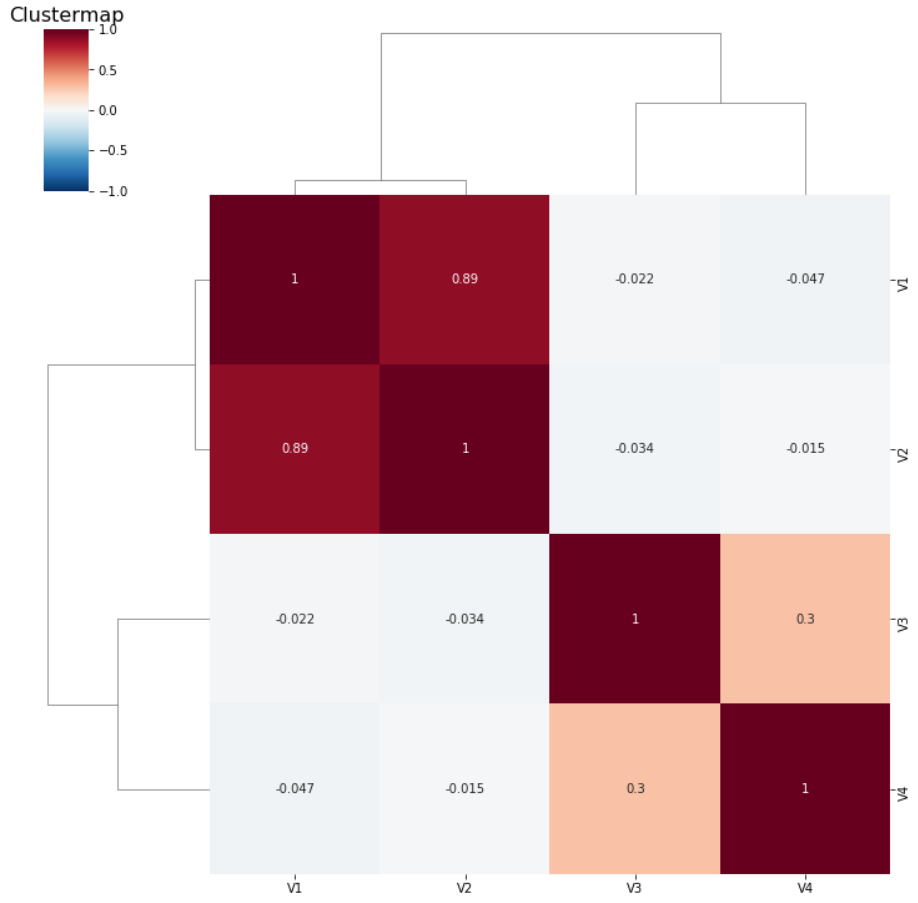
```
Text(0.5, 1.0, 'Correlation Matrix')
```



There is also a neat type of heatmap that will also reorganize your variable based on clustering.

```
sns.clustermap(corr, square=True, annot=True, cmap='RdBu_r', vmin=-1, vmax=1)
plt.title('Clustermap', fontsize=16)
```

```
Text(0.5, 1.0, 'Clustermap')
```



Pandas

We introduced `Pandas` in a previous tutorial. It can also call matplotlib to quickly generate plots.

We will use the same dataset used in that tutorial to generate some plots using `pd.Dataframe`.

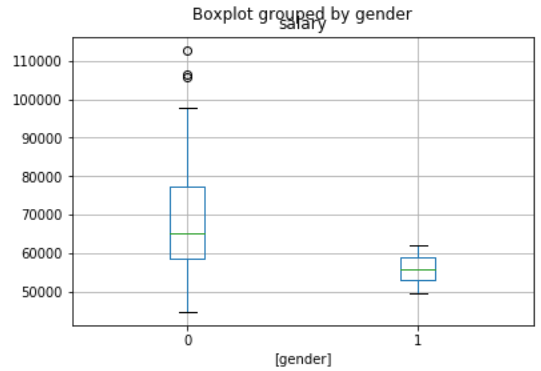
```
df = pd.read_csv('../../data/salary.csv', sep = ',', header='infer')
df = df.dropna()
df = df[df['gender']!=2]
```

```
# key: We use the departm as the grouping factor.
key = df['departm']

# Let's create an anonmyous function for calculating zscores using lambda:
# We want to standardize salary for each department.
zscore = lambda x: (x - x.mean()) / x.std()

# Now let's calculate zscores separately within each department
transformed = df.groupby(key).transform(zscore)
df['salary_in_departm'] = transformed['salary']
```

Now we have `salary_in_departm` column showing standardized salary per department.

```
df[['salary','gender']].boxplot(by='gender')
```

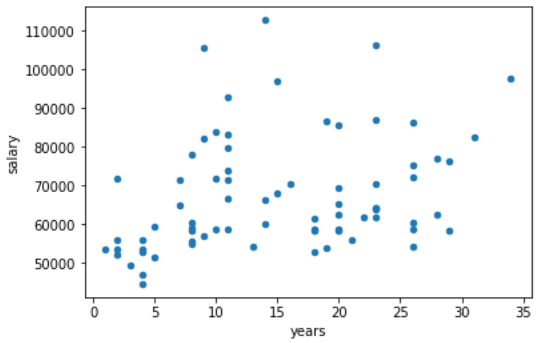
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1d97f320>
```



Scatterplot

```
df[['salary', 'years']].plot(kind='scatter', x='years', y='salary')
```

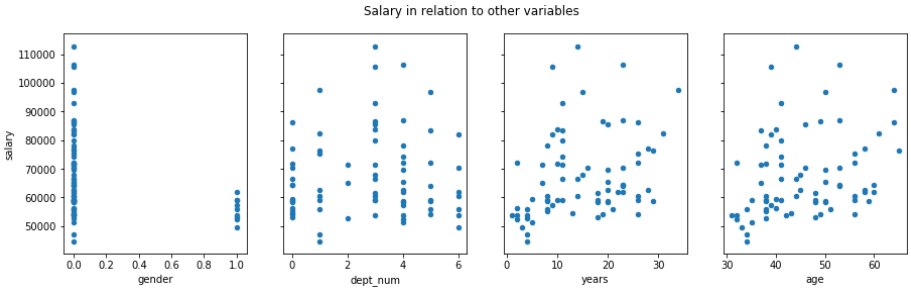
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1dc9df28>
```



Now plot all four categories

```
f, axs = plt.subplots(1, 4, sharey=True)
f.suptitle('Salary in relation to other variables')
df.plot(kind='scatter', x='gender', y='salary', ax=axs[0], figsize=(15, 4))
df.plot(kind='scatter', x='dept_num', y='salary', ax=axs[1])
df.plot(kind='scatter', x='years', y='salary', ax=axs[2])
df.plot(kind='scatter', x='age', y='salary', ax=axs[3])
```

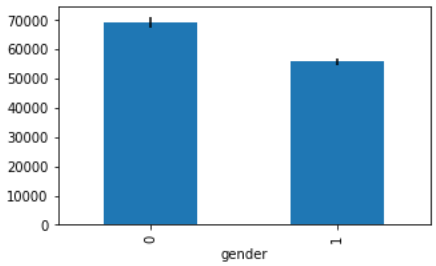
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1dedb128>
```



The problem is that it treats department as a continuous variable.

Generating bar - errorbar plots in Pandas

```
means = df.groupby('gender').mean()['salary']
errors = df.groupby('gender').std()['salary'] / np.sqrt(df.groupby('gender').count()
['salary'])
ax = means.plot.bar(yerr=errors,figsize=(5,3))
```



Interactive Plots

Interactive data visualizations are an exciting development and are likely to continue to grow in popularity with the rapid developments in web frontend frameworks. Covering these libraries is beyond the scope of this tutorial, but I highly encourage you to check them out. Some of them are surprisingly easy to use and make exploring your data and sharing your results much more fun.

It is possible to add some basic interactivity to the plotting libraries covered in this tutorial in jupyter notebooks with ipywidgets. You will see a few examples of this in other tutorials.

Plotly

Plotly is an graphing library to make interactive plots.

Bokeh

Bokeh is an interactive visualization library

Altair

Altair is a declarative statistical visualization library for Python based on Vega

Exercises

The following exercises uses the dataset “salary_exercise.csv” adapted from material available here

These are the salary data used in Weisberg’s book, consisting of observations on six variables for 52 tenure-track professors in a small college. The variables are:

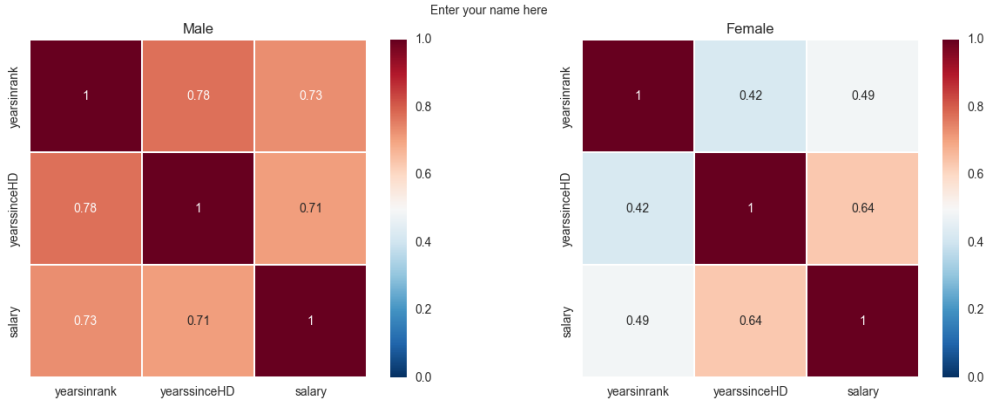
- sx = Sex, coded 1 for female and 0 for male
- rk = Rank, coded
- 1 for assistant professor,
- 2 for associate professor, and

- 3 for full professor
- yr = Number of years in current rank
- dg = Highest degree, coded 1 if doctorate, 0 if masters
- yd = Number of years since highest degree was earned
- sl = Academic year salary, in dollars.

Reference: S. Weisberg (1985). Applied Linear Regression, Second Edition. New York: John Wiley and Sons. Page 194.

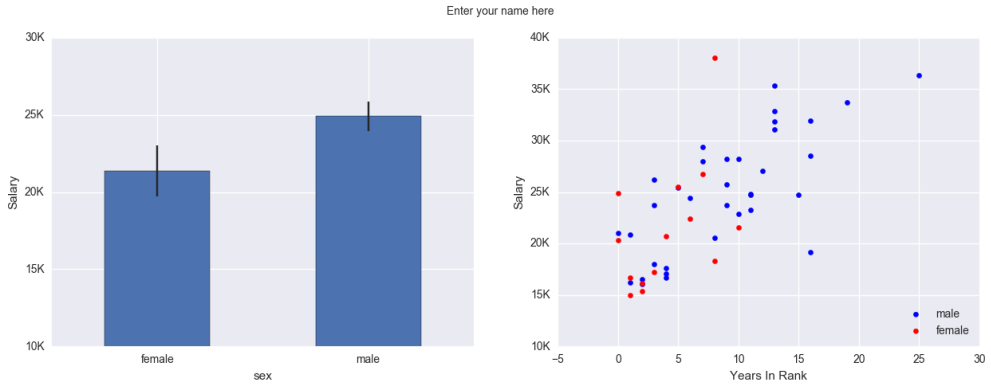
Exercise 1

Recreate the plot shown in figure.
On the left is a correlation of all parameters of only the male professors.
On the right is the same but only for female professors.
The colormap code used is `RdBu_r`. Read the Docstrings on sns.heatmap or search the internet to figure out how to change the colormap, scale the colorbar, and create square line boundaries.
Place titles for each plot as shown, and your name as the main title.



Exercise 2

Recreate the following plot from the salary_exercise.csv dataset.
Create a 1 x 2 subplot.
On the left is a bar-errorbar of salary per gender.
On the right is a scatterplot of salary on y-axis and years in rank on the x-axis.
Set the axis limits as shown in the picture and modify their lables.
Add axis label names.
Add a legend for the scatterplot and place it at a bottom-right location.
Add your name as the main title of the plot.



Glossary

Written by Luke Chang

Throughout this course we will use a variety of different functions available in the base Python library, but also many other libraries in the scientific computing stack. Here we provide a list of all of the functions that are used across the various notebooks. It can be a helpful reference when you are learning Python about the types of things you can do with various packages. Remember you can always view the docstrings for any function by adding a `?` to the end of the function name.

Jupyter Cell Magic

Magics are specific to and provided by the IPython kernel. Whether Magics are available on a kernel is a decision that is made by the kernel developer on a per-kernel basis. To work properly, Magics must use a syntax element which is not valid in the underlying language. For example, the IPython kernel uses the `%` syntax element for Magics as `%` is not a valid unary operator in Python. However, `%` might have meaning in other languages.

%conda: Run the conda package manager within the current kernel.

%debug: Activate the interactive debugger. This magic command support two ways of activating debugger. One is to activate debugger before executing code. This way, you can set a break point, to step through the code from the point. You can use this mode by giving statements to execute and optionally a breakpoint. The other one is to activate debugger in post-mortem mode. You can activate this mode simply running %debug without any argument. If an exception has just occurred, this lets you inspect its stack frames interactively. Note that this will always work only on the last traceback that occurred, so you must call this quickly after an exception that you wish to inspect has fired, because if another one occurs, it clobbers the previous one.

%matplotlib: Set up matplotlib to work interactively. Example: `%matplotlib inline`

This function lets you activate matplotlib interactive support at any point during an IPython session. It does not import anything into the interactive namespace.

%timeit: Time execution of a Python statement or expression using the timeit module. This function can be used both as a line and cell magic

!: Shell execute - run shell command and capture output (!! is short-hand). Example: `!pip`.

Base Python Functions

These functions are all bundled with Python

any: Test if any of the elements are true.

bool: Cast as boolean type

dict: Cast as dictionary type

enumerate: Return an enumerate object. iterable must be a sequence, an iterator, or some other object which supports iteration. The **next**() method of the iterator returned by enumerate() returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over iterable.

float: Return a floating point number constructed from a number or string x.

import: Import python module into namespace.

int: Cast as integer type

len: Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

glob.glob: The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the `os.scandir()` and `fnmatch.fnmatch()` functions in concert, and not by actually invoking a subshell.

list: Cast as list type

max: Return the largest item in an iterable or the largest of two or more arguments.

min: Return the smallest item in an iterable or the smallest of two or more arguments.

os.path.basename: Return the base name of pathname path. This is the second element of the pair returned by passing path to the function split(). Note that the result of this function is different from the Unix basename program; where basename for '/foo/bar/' returns 'bar', the basename() function returns an empty string ('').

os.path.join: Join one or more path components intelligently. The return value is the concatenation of path and any members of paths with exactly one directory separator (os.sep) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

print: Print strings. Recommend using f-strings formatting. Example, `print(f'Results: {variable}')`.

pwd: Print current working directory

sorted: Return a new sorted list from the items in iterable.

str: For more information on static methods, see The standard type hierarchy.

range: Rather than being a function, range is actually an immutable sequence type, as documented in Ranges and Sequence Types — list, tuple, range.

tuple: Cast as tuple type

type: Return the type of the object.

zip: Make an iterator that aggregates elements from each of the iterables.

Pandas

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

```
import pandas as pd
```

---

pd.read_csv: Read a comma-separated values (csv) file into DataFrame.

pd.concat: Concatenate pandas objects along a particular axis with optional set logic along the other axes.

pd.DataFrame.isnull: Detect missing values.

pd.DataFrame.mean: Return the mean of the values for the requested axis.

pd.DataFrame.std: Return sample standard deviation over requested axis.

pd.DataFrame.plot: Plot data using matplotlib

pd.DataFrame.map: Map values of Series according to input correspondence.

pd.DataFrame.groupby: Group DataFrame or Series using a mapper or by a Series of columns.

pd.DataFrame.fillna: Fill NA/NaN values using the specified method.

pd.DataFrame.replace: Replace values given in to_replace with value.

NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

```
import numpy as np
```

np.arange: Return evenly spaced values within a given interval.

np.array: Create an array

np.convolve: Returns the discrete, linear convolution of two one-dimensional sequences.

np.cos: Trigonometric cosine element-wise.

np.diag: Extract a diagonal or construct a diagonal array.

np.diag_indices: Return the indices to access the main diagonal of an array.

np.dot: Dot product of two arrays.

np.exp: Calculate the exponential of all elements in the input array.

np.fft.fft: Compute the one-dimensional discrete Fourier Transform.

np.fft.ifft: Compute the one-dimensional inverse discrete Fourier Transform.

np.hstack: Stack arrays in sequence horizontally (column wise).

np.linalg.pinv: Compute the (Moore-Penrose) pseudo-inverse of a matrix.

np.mean: Compute the arithmetic mean along the specified axis.

np.nan: IEEE 754 floating point representation of Not a Number (NaN).

np.ones: Return a new array of given shape and type, filled with ones.

np.pi: Return pi 3.14159265358979323846264 33...

np.random.randint: Return random integers from low (inclusive) to high (exclusive).

np.random.randn: Return a sample (or samples) from the “standard normal” distribution.

np.real: Return the real part of the complex argument.

np.sin: Trigonometric sine, element-wise.

np.sqrt: Return the non-negative square-root of an array, element-wise.

np.squeeze: Remove single-dimensional entries from the shape of an array.

np.std: Compute the standard deviation along the specified axis.

np.vstack: Stack arrays in sequence vertically (row wise).

np.zeros: Return a new array of given shape and type, filled with zeros.

SciPy

SciPy is one of the core packages that make up the SciPy stack. It provides many user-friendly and efficient numerical routines, such as routines for numerical integration, interpolation, optimization, linear algebra, and statistics.

---

scipy.stats.binom: A binomial discrete random variable.

scipy.signal.butter: Butterworth digital and analog filter design.

scipy.signal.filtfilt: Apply a digital filter forward and backward to a signal.

scipy.signal.freqz: Compute the frequency response of a digital filter.

scipy.signal.sosfreqz: Compute the frequency response of a digital filter in SOS format.

scipy.stats.ttest_1samp: Calculate the T-test for the mean of ONE group of scores.

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

```
import matplotlib.pyplot as plt
```

---

plt.bar: Make a bar plot.

plt.figure: Create a new figure.

plt.hist: Plot a histogram.

plt.imshow: Display an image, i.e. data on a 2D regular raster.

plt.legend: Place a legend on the axes.

plt.savefig: Save the current figure.

plt.scatter: A scatter plot of y vs x with varying marker size and/or color.

plt.subplots: Create a figure and a set of subplots.

plt.tight_layout: Automatically adjust subplot parameters to give specified padding.

ax.axvline: Add a vertical line across the axes.

ax.set_xlabel: Set the label for the x-axis.

ax.set_xlim: Set the x-axis view limits.

ax.set_xticklabels: Set the x-tick labels with list of string labels.

ax.set_ylim: Set the y-axis view limits.

ax.set_yticklabels: Set the y-tick labels with list of string labels.

ax.set_ylabel: Set the label for the y-axis.

ax.set_title: Set a title for the axes.

Seaborn

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

```
import seaborn as sns
```

---

sns.heatmap: Plot rectangular data as a color-encoded matrix.

sns.catplot: Figure-level interface for drawing categorical plots onto a FacetGrid.

sns.jointplot: Draw a plot of two variables with bivariate and univariate graphs.

sns.regplot: Plot data and a linear regression model fit.

scikit-learn

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

---

sklearn.metrics.pairwise_distances: This method takes either a vector array or a distance matrix, and returns a distance matrix. If the input is a vector array, the distances are computed. If the input is a distances matrix, it is returned instead.

sklearn.metrics.balanced_accuracy_score: Compute the balanced accuracy. The balanced accuracy in binary and multiclass classification problems to deal with imbalanced datasets. It is defined as the average of recall obtained on each class.

networkx

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

```
import networkx as nx
```

---

nx.draw_kamada_kawai: Draw the graph G with a Kamada-Kawai force-directed layout.

nx.degree: Return the degree of a node or nodes. The node degree is the number of edges adjacent to that node.

NiBabel

nibabel is a package to help Read / write access to some common neuroimaging file formats, including: ANALYZE (plain, SPM99, SPM2 and later), GIFTI, NIfTI1, NIfTI2, CIFTI-2, MINC1, MINC2, AFNI BRIK/HEAD, MGH and ECAT as well as Philips PAR/REC. We can read and write FreeSurfer geometry, annotation and morphometry files. There is some very limited support for DICOM. NiBabel is the successor of PyNIfTI.

The various image format classes give full or selective access to header (meta) information and access to the image data is made available via NumPy arrays.

```
import nibabel as nib
```

---

nib.load: Load file given filename, guessing at file type

nib.save: Save an image to file adapting format to filename

data.get_data: Return image data from image with any necessary scaling applied

data.get_shape: Return shape for image

data.header: The header of an image contains the image metadata. The information in the header will differ between different image formats. For example, the header information for a NIfTI1 format file differs from the header information for a MINC format file.

data.affine: homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, obj.affine also returns None, and the affine as written to disk will depend on the file format.

NiLearn

nilearn is a Python module for fast and easy statistical learning on NeuroImaging data.

It leverages the scikit-learn Python toolbox for multivariate statistics with applications such as predictive modelling, classification, decoding, or connectivity analysis.

---

nilearn.plotting.plot_anat: Plot cuts of an anatomical image (by default 3 cuts: Frontal, Axial, and Lateral)

nilearn.plotting.view_img:Interactive html viewer of a statistical map, with optional background

nilearn.plotting.plot_glass_brain: Plot 2d projections of an ROI/mask image (by default 3 projections: Frontal, Axial, and Lateral). The brain glass schematics are added on top of the image.

nilearn.plotting.plot_stat_map: Plot cuts of an ROI/mask image (by default 3 cuts: Frontal, Axial, and Lateral)

nltools

NLTools is a Python package for analyzing neuroimaging data. It is the analysis engine powering neuro-learn There are tools to perform data manipulation and analyses such as univariate GLMs, predictive multivariate modeling, and representational similarity analyses.

Data Classes

Adjacency

Adjacency is a class to represent Adjacency matrices as a vector rather than a 2-dimensional matrix. This makes it easier to perform data manipulation and analyses. This tool is particularly useful for performing Representational Similarity Analyses.

Adjacency.distance: Calculate distance between images within an Adjacency() instance.

Adjacency.distance_to_similarity: Convert distance matrix to similarity matrix

Adjacency.plot: Create Heatmap of Adjacency Matrix

Adjacency.plot_mds: Plot Multidimensional Scaling

Adjacency.to_graph: Convert Adjacency into networkx graph. only works on single_matrix for now.

Brain_Data

Brain_Data is a class to represent neuroimaging data in python as a vector rather than a 3-dimensional matrix.This makes it easier to perform data manipulation and analyses. This is the main tool for working with neuroimaging data.

Brain_Data.append: Append data to Brain_Data instance

apply_mask: Mask Brain_Data instance

Brain_Data.copy: Create a copy of a Brain_Data instance.

Brain_Data.decompose: Decompose Brain_Data object

Brain_Data.distance: Calculate distance between images within a Brain_Data() instance.

Brain_Data.extract_roi: Extract activity from mask

Brain_Data.find_spikes: Function to identify spikes from Time Series Data

Brain_Data.iplot: Create an interactive brain viewer for the current brain data instance.

Brain_Data.mean: Get mean of each voxel across images.

Brain_Data.plot: Create a quick plot of self.data. Will plot each image separately

Brain_Data.predict: Run prediction

Brain_Data.regress: Run a mass-univariate regression across voxels. Three types of regressions can be run: 1) Standard OLS (default) 2) Robust OLS (heteroscedasticty and/or auto-correlation robust errors), i.e. OLS with “sandwich estimators” 3) ARMA (auto-regressive and moving-average lags = 1 by default; experimental)

Brain_Data.shape: Get images by voxels shape.

Brain_Data.similarity: Calculate similarity of Brain_Data() instance with single Brain_Data or Nibabel image

Brain_Data.smooth: Apply spatial smoothing using nilearn smooth_img()

Brain_Data.std: Get standard deviation of each voxel across images.

Brain_Data.threshold: Threshold Brain_Data instance.

Brain_Data.to_nifti: Convert Brain_Data Instance into Nifti Object

Brain_Data.ttest: Calculate one sample t-test across each voxel (two-sided)

Brain_Data.write: Write out Brain_Data object to Nifti or HDF5 File.

Design_Matrix

Design_Matrix is a class to represent design matrices with special methods for data processing (e.g. convolution, upsampling, downsampling) and also intelligent and flexible and intelligent appending (e.g. auto-matically keep certain columns or polynomial terms separated during concatentation). It plays nicely with Brain_Data and can be used to build an experimental design to pass to Brain_Data's X attribute. It is essentially an enhanced pandas df, with extra attributes and methods. Methods always return a new design matrix instance (copy). Column names are always string types. Inherits most methods on pandas DataFrames.

Design_Matrix.add_dct_basis: Adds unit scaled cosine basis functions to Design_Matrix columns, based on spm-style discrete cosine transform for use in high-pass filtering. Does not add intercept/constant. Care is recommended if using this along with .add_poly(), as some columns will be highly-correlated.

Design_Matrix.add_poly: Add nth order Legendre polynomial terms as columns to design matrix. Good for adding constant/intercept to model (order = 0) and accounting for slow-frequency nuisance artifacts e.g. linear, quadratic, etc drifts. Care is recommended when using this with .add_dct_basis() as some columns will be highly correlated.

Design_Matrix.clean: Method to fill NaNs in Design Matrix and remove duplicate columns based on data values, NOT names. Columns are dropped if they are correlated >= the requested threshold (default = .95). In this case, only the first instance of that column will be retained and all others will be dropped.

Design_Matrix.convolve: Perform convolution using an arbitrary function.

Design_Matrix.heatmap: Visualize Design Matrix spm style. Use .plot() for typical pandas plotting functionality. Can pass optional keyword args to seaborn heatmap.

Design_Matrix.head: This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Design_Matrix.info: Print a concise summary of a DataFrame.

Design_Matrix.vif: Compute variance inflation factor amongst columns of design matrix, ignoring polynomial terms. Much faster that statsmodel and more reliable too. Uses the same method as Matlab and R (diagonal elements of the inverted correlation matrix).

Design_Matrix.zscore: nltools.stats.downsample, but ensures that returned object is a design matrix.

Statistics Functions

stats.fdr: Determine FDR threshold given a p value array and desired false discovery rate q.

stats.find_spikes: Function to identify spikes from fMRI Time Series Data

stats.fisher_r_to_z: Use Fisher transformation to convert correlation to z score

stats.one_sample_permutation: One sample permutation test using randomization.

stats.threshold: Threshold test image by p-value from p image

stats.regress: This is a flexible function to run several types of regression models provided X and Y numpy arrays. Y can be a 1d numpy array or 2d numpy array. In the latter case, results will be output with shape 1 x Y.shape[1], in other words fitting a separate regression model to each column of Y.

stats.zscore: zscore every column in a pandas dataframe or series.

Miscellaneous Functions

SimulateGrid: A class to simulate signal and noise within 2D grid. Need to update link to nltools documentation once it is built.

external.hrf.glover_hrf: Implementation of the Glover hemodynamic response function (HRF) model.

datasets.fetch_pain: Download and loads pain dataset from neurovault

datasets.fetch_localizer: Download and load Brainomics Localizer dataset (94 subjects).

Introduction to Neuroimaging

Measuring in-vivo brain activity from humans is an extraordinary feat. What is Neuroimaging? and What can we learn about the brain using this technique?

In this section, we will provide an overview of the course and a very introductory overview of the different types neuroimaging and a few examples of different types of things we can do with BOLD fMRI.

Lecture

The lecture for this section can be viewed here.

Measurement and Signal

Measuring in-vivo brain activity from humans is an extraordinary feat. How do scanners work? and what exactly are we measuring?

In this section, we will provide a very introductory overview of the basics of MR Physics and the physiology underlying the signal we are measuring with BOLD fMRI.

Lecture

The lecture for this section can be viewed here.

Readings

Please read Chapter 7 of the Huettel, Song, & McCarthy textbook on the BOLD signal

Videos

There are many fantastic lectures on this topic. We recommend watching the short videos by Tor Wager and Martin Lindquist from their Principles of fMRI Coursera class.

- Basic MR Physics
- Image Formation
- K-Space
- Signal & BOLD Physiology

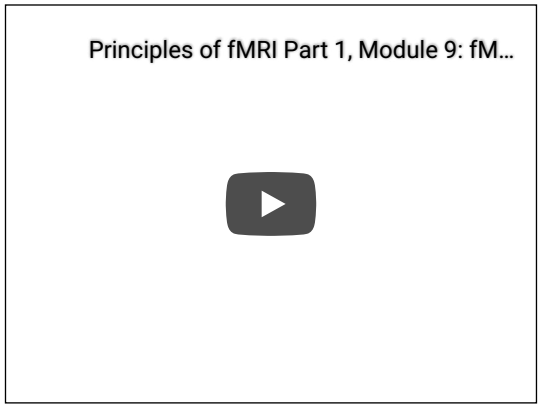
Separating Signal From Noise With ICA

Written by Luke Chang

In this tutorial we will use ICA to explore which signals in our imaging data might be real signal or artifacts.

For a brief overview of types of artifacts that might be present in your data, I recommend watching this video by Tor Wager and Martin Lindquist.

```
from IPython.display import YouTubeVideo

YouTubeVideo('7Kk_RsGycHs')
```



Preprocessing Data

To run this tutorial, you must have run preprocessing on at least one participant. *If you are in Psych60, this has already been done for you and you can just skip to* ***Loading Data***. If you reading this online, then I recommend preprocessing your data with fmriprep, which is a robust, but opinionated automated preprocessing pipeline developed by Russ Poldrack's group at Stanford University. The developer's have made a number of choices about how to preprocess your fMRI data using best practices and have created an automated pipeline using multiple software packages that are all distributed via a docker container.

In theory, this is extraodinarily straightforward to run:

- 1. Install Docker and download image

```
docker pull poldracklab/fmriprep:<latest-version>
```

- 1. Run a single command in the terminal specifying the location of the data, the location of the output, the participant id, and a few specific flags depending on specific details of how you want to run the preprocessing.

```
fmriprep-docker /Users/lukechang/Dropbox/Dartbrains/Data/localizer
/Users/lukechang/Dropbox/Dartbrains/Data/preproc participant --participant_label sub-S01 --
write-graph --fs-no-reconall --notrack --fs-license-file
~/Dropbox/Dartbrains/License/license.txt --work-dir
/Users/lukechang/Dropbox/Dartbrains/Data/work
```

In practice, it's alway a little bit finicky to get everything set up on a particular system. Sometimes you might run into issues with a specific missing file like the freesurfer license even if you're not using it. You might also run into issues with the format of the data that might have some conflicts with the bids-validator. In our experience, there is always some frustrations getting this to work, but it's very nice once it's done.

Loading Data

Ok, once you've finished preprocessing some of your data with fmriprep, we can load a subject and run an ICA to explore signals that are present. Since we have completed preprocessing, our data should be realigned and also normalized to MNI stereotactic space. We will use the nltools package to work with this data in python.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.data import Brain_Data
from nltools.plotting import component_viewer

base_dir = '../data/localizer/derivatives/preproc/fmriprep'
base_dir = '/Users/lukechang/Dropbox/Dartbrains/Data/preproc/fmriprep'
sub = 'S01'

data = Brain_Data(os.path.join(base_dir, f'sub-{sub}','func', f'sub-{sub}_task-
localizer_space-MNI152NLin2009cAsym_desc-preproc_bold.nii.gz'))
```

More Preprocessing

Even though, we have technically already run most of the preprocessing there are a couple of more steps that will help make the ICA cleaner.

First, we will run a high pass filter to remove any low frequency scanner drift. We will pick a fairly arbitrary filter size of 0.0078hz (1/128s). We will also run spatial smoothing with a 6mm FWHM gaussian kernel to increase a signal to noise ratio at each voxel. These steps are very easy to run using nltools after the data has been loaded.

```
data = data.filter(sampling_freq=1/2.4, high_pass=1/128)

data = data.smooth(6)
```

Independent Component Analysis (ICA)

Ok, we are finally ready to run an ICA analysis on our data.

ICA attempts to perform blind source separation by decomposing a multivariate signal into additive subcomponents that are maximally independent.

We will be using the `decompose()` method on our `Brain_Data` instance. This runs the FastICA algorithm implemented by scikit-learn. You can choose whether you want to run spatial ICA by setting `axis='voxels` or temporal ICA by setting `axis='images'`. We also recommend running the whitening flat `whiten=True`. By default `decompose` will estimate the maximum components that are possible given the data. We recommend using a completely arbitrary heuristic of 20-30 components.

```
tr = 2.4
output = data.decompose(algorithm='ica', n_components=30, axis='images', whiten=True)
```

Viewing Components

We will use the interactive `component_viewer` from nltools to explore the results of the analysis. This viewer uses ipywidgets to select the `Component` to view and also the threshold. You can manually enter a component number to view or scroll up and down.

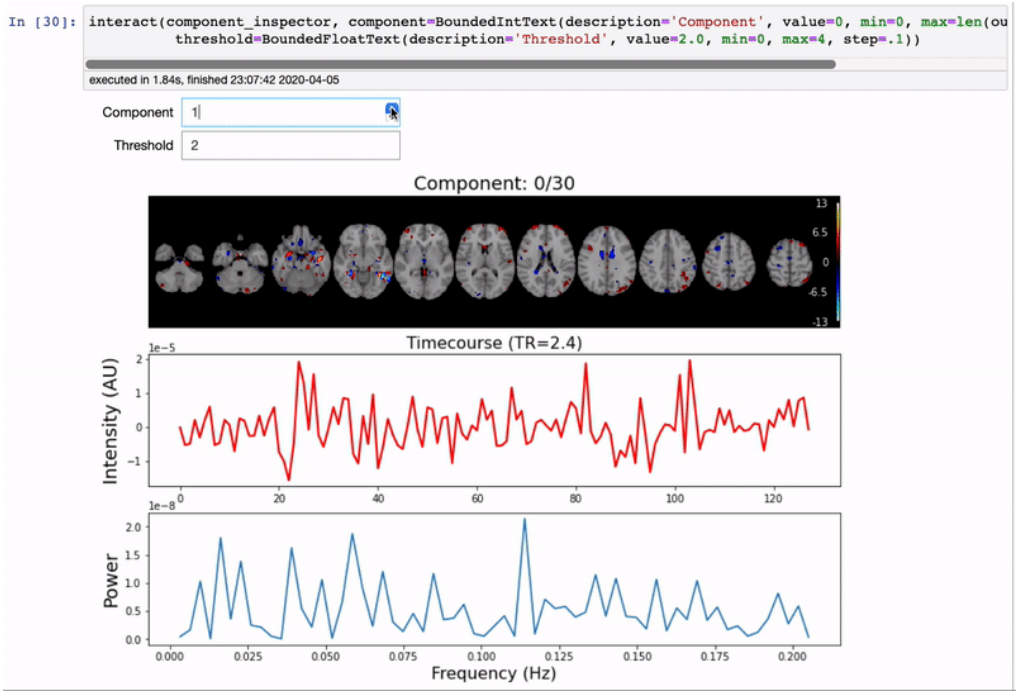
Components have been standardized, this allows us to threshold the brain in terms of standard deviations. For example, the default threshold of 2.0, means that any voxel that loads on the component greater or less than 2 standard deviations will be overlaid on the standard brain. You can play with different thresholds to be more or less inclusive - a threshold of 0 will overlay all of the voxels. If you play with any of the numbers, make sure you press tab to update the plot.

The second plot is the time course of the voxels that load on the component. The x-axis is in TRs, which for this dataset is 2.4 sec.

The third plot is the powerspectrum of the timecourse. There is not a large range of possible values as we can only observe signals at the nyquist frequency, which is half of our sampling frequency of 1/2.4s (approximately 0.21hz) to a lower bound of 0.0078hz based on our high pass filter. There might be systematic oscillatory signals. Remember, that signals that oscillate a faster frequency than the nyquist frequency will be aliased. This includes physiological artifacts such as respiration and cardiac signals.

It is important to note that ICA cannot resolve the sign of the component. So make sure you consider signals that are positive as well as negative.

```
component_viewer(output, tr=2.4)
```



Exercises

For this tutorial, try to guess which components are signal and which are noise. Also, be sure to label the type of noise you think you might be seeing (e.g., head motion, scanner spikes, cardiac, respiration, etc.) Do this for subjects `s01` and `s02`.

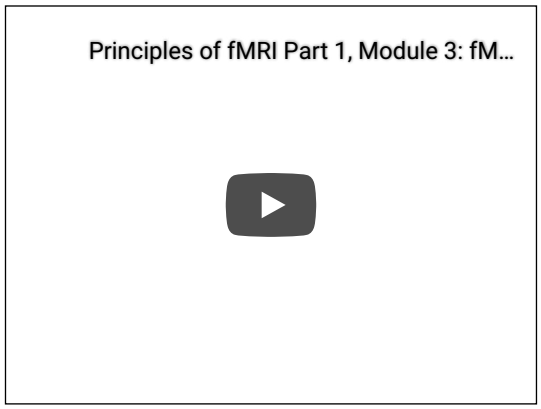
What features do you think are important to consider when making this judgment? Does the spatial map provide any useful information? What about the timecourse of the component? Does it map on to the plausible timecourse of the task.What about the power spectrum?

Introduction to Neuroimaging Data

In this tutorial we will learn the basics of the organization of data folders, and how to load, plot, and manipulate neuroimaging data in Python.

To introduce the basics of fMRI data structures, watch this short video by Martin Lindquist.

```
from IPython.display import YouTubeVideo

YouTubeVideo('0uRdQJMU5ro')
```



Software Packages

There are many different software packages to analyze neuroimaging data. Most of them are open source and free to use (with the exception of BrainVoyager). The most popular ones (SPM, FSL, & AFNI) have been around a long time and are where many new methods are developed and distributed. These packages have focused on implementing what they believe are the best statistical methods, ease of use, and computational efficiency. They have very large user bases so many bugs have been identified and fixed over the years. There are also lots of publicly available documentation, listserves, and online tutorials, which makes it very easy to get started using these tools.

There are also many more boutique packages that focus on specific types of preprocessing step and analyses such as spatial normalization with ANTs, connectivity analyses with the conn-toolbox, representational similarity analyses with the rsaToolbox, and prediction/classification with pyMVPA.

Many packages have been developed within proprietary software such as Matlab (e.g., SPM, Conn, RSAToolbox, etc). Unfortunately, this requires that your university has site license for Matlab and many individual add-on toolboxes. If you are not affiliated with a University, you may have to pay for Matlab, which can be fairly expensive. There are free alternatives such as octave, but octave does not include many of the add-on toolboxes offered by matlab that may be required for a specific package. Because of this restrictive licensing, it is difficult to run matlab on cloud computing servers and to use with free online courses such as dartbrains. Other packages have been written in C/C++/C# and need to be compiled to run on your specific computer and operating system. While these tools are typically highly computationally efficient, it can sometimes be challenging to get them to install and work on specific computers and operating systems.

There has been a growing trend to adopt the open source Python framework in the data science and scientific computing communities, which has lead to an explosion in the number of new packages available for statistics, visualization, machine learning, and web development. pyMVPA was an early leader in this trend, and there are many great tools that are being actively developed such as nilearn, brainiak, neurosynth, nipype, fmriprep, and many more. One exciting thing is that these newer developments have built on the expertise of decades of experience with imaging analyses, and leverage changes in high performance computing. There is also a very tight integration with many cutting edge developments in adjacent communities such as machine learning with scikit-learn, tensorflow, and pytorch, which has made new types of analyses much more accessible to the neuroimaging community. There has also been an influx of younger contributors with software development expertise. You might be surprised to know that many of the popular tools being used had core contributors originating from the neuroimaging community (e.g., scikit-learn, seaborn, and many more).

For this course, I have chosen to focus on tools developed in Python as it is an easy to learn programming language, has excellent tools, works well on distributed computing systems, has great ways to disseminate information (e.g., jupyter notebooks, jupyter-book, etc), and is free! If you are just getting started, I would spend some time working with NiLearn and Brainiak, which have a lot of functionality, are very well tested, are reasonably computationally efficient, and most importantly have lots of documentation and tutorials to get started.

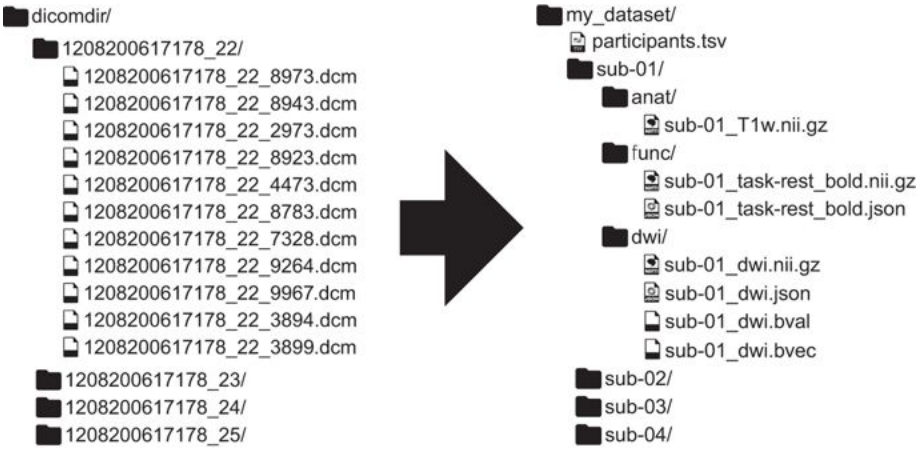
We will be using many packages throughout the course such as PyBids to navigate neuroimaging datasets, fmriprep to perform preprocessing, and nltools, which is a package developed in my lab, to do basic data manipulation and analysis. NLtools is built using many other toolboxes such as nibabel and nilearn, and we will also be using these frequently throughout the course.

BIDS: Brain Imaging Dataset Specification

Recently, there has been growing interest to share datasets across labs and even on public repositories such as openneuro. In order to make this a successful enterprise, it is necessary to have some standards in how the data are named and organized. Historically, each lab has used their own idiosyncratic conventions, which can make it difficult for outsiders to analyze. In the past few years, there have been heroic efforts by the neuroimaging community to create a standardized file organization and naming practices. This specification is called **BIDS** for Brain Imaging Dataset Specification.

As you can imagine, individuals have their own distinct method of organizing their files. Think about how you keep track of your files on your personal laptop (versus your friend). This may be okay in the personal realm, but in science, it's best if anyone (especially yourself 6 months from now!) can follow your work and know *which* files mean *what* by their titles.

Here's an example of non-Bids versus BIDS dataset found in this paper:



Here are a few major differences between the two datasets:

1. In BIDS, files are in nifti format (not dicoms).
2. In BIDS, scans are broken up into separate folders by type of scan(functional versus anatomical versus diffusion weighted) for each subject.
3. In BIDS, JSON files are included that contain descriptive information about the scans (e.g., acquisition parameters)

Not only can using this specification be useful within labs to have a set way of structuring data, but it can also be useful when collaborating across labs, developing and utilizing software, and publishing data.

In addition, because this is a consistent format, it is possible to have a python package to make it easy to query a dataset. We recommend using pybids.

The dataset we will be working with has already been converted to the BIDS format (see download localizer tutorial).

You may need to install pybids to query the BIDS datasets using following command `!pip install pybids`.

The `BIDSLayout`

Pybids is a package to help query and navigate a neuroimaging dataset that is in the BIDs format. At the core of pybids is the `BIDSLayout` object. A `BIDSLayout` is a lightweight Python class that represents a BIDS project file tree and provides a variety of helpful methods for querying and manipulating BIDS files. While the BIDSLayout initializer has a large number of arguments you can use to control the way files are indexed and accessed, you will most commonly initialize a BIDSLayout by passing in the BIDS dataset root location as a single argument.

Notice we are setting `derivatives=True`. This means the layout will also index the derivatives sub folder, which might contain preprocessed data, analyses, or other user generated files.

```
from bids import BIDSLayout, BIDSValidator
import os

data_dir = '/Users/lukechang/Dropbox/Dartbrains/data/Localizer'
layout = BIDSLayout(data_dir, derivatives=True)
layout
```

```
BIDS Layout: ...pbox/Dartbrains/data/Localizer | Subjects: 94 | Sessions: 0 | Runs: 0
```

When we initialize a BIDSLayout, all of the files and metadata found under the specified root folder are indexed. This can take a few seconds (or, for very large datasets, a minute or two). Once initialization is complete, we can start querying the BIDSLayout in various ways. The main query method is `.get()`. If we call `.get()` with no additional arguments, we get back a list of all the BIDS files in our dataset.

Let's return the first 10 files

```
layout.get()[:10]
```

```
[<BIDSJSONFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/dataset_description.json
'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/.DS
_Store'>,
 <BIDSJSONFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/dat
aset_description.json'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/log
s/CITATION.bib'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/log
s/CITATION.html'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/log
s/CITATION.md'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/log
s/CITATION.tex'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/sub
-S01.html'>,
 <BIDSFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/sub
-S01/.DS_Store'>,
 <BIDSJSONFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/derivatives/fmriprep/sub
-S01/anat/sub-S01_desc-brain_mask.json'>]
```

As you can see, just a generic `.get()` call gives us *all* of the files. We will definitely want to be a bit more specific. We can specify the type of data we would like to query. For example, suppose we want to return the first 10 subject ids.

```
layout.get(target='subject', return_type='id')[:10]
```

```
['S01', 'S02', 'S03', 'S04', 'S05', 'S06', 'S07', 'S08', 'S09', 'S10']
```

Or perhaps, we would like to get the file names for the raw bold functional nifti images for the first 10 subjects. We can filter files in the `raw` or `derivatives`, using `scope` keyword.`scope='raw'`, to only query raw bold nifti files.

```
layout.get(target='subject', scope='raw', suffix='bold', return_type='file')[:10]
```

```
['/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S01/func/sub-S01_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S02/func/sub-S02_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S03/func/sub-S03_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S04/func/sub-S04_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S05/func/sub-S05_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S06/func/sub-S06_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S07/func/sub-S07_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S08/func/sub-S08_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S09/func/sub-S09_task-
localizer_bold.nii.gz',
 '/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-S10/func/sub-S10_task-
localizer_bold.nii.gz']
```

When you call .get() on a BIDSLayout, the default returned values are objects of class BIDSFile. A BIDSFile is a lightweight container for individual files in a BIDS dataset.

Here are some of the attributes and methods available to us in a BIDSFile (note that some of these are only available for certain subclasses of BIDSFile; e.g., you can't call get_image() on a BIDSFile that doesn't correspond to an image file!):

- .path: The full path of the associated file
- .filename: The associated file's filename (without directory)
- .dirname: The directory containing the file

- .get_entities(): Returns information about entities associated with this BIDSFile (optionally including metadata)
- .get_image(): Returns the file contents as a nibabel image (only works for image files)
- .get_df(): Get file contents as a pandas DataFrame (only works for TSV files)
- .get_metadata(): Returns a dictionary of all metadata found in associated JSON files
- .get_associations(): Returns a list of all files associated with this one in some way

Let's explore the first file in a little more detail.

```
f = layout.get()[0]
f
```

```
<BIDSJSONFile
filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/dataset_description.json
'>
```

If we wanted to get the path of the file, we can use `.path`.

```
f.path
```

```
'/Users/lukechang/Dropbox/Dartbrains/Data/localizer/dataset_description.json'
```

Suppose we were interested in getting a list of tasks included in the dataset.

```
layout.get_task()
```

```
['localizer']
```

We can query all of the files associated with this task.

```
layout.get(task='localizer', suffix='bold', scope='raw')[:10]
```

```
[<BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S01/func/sub-S01_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S02/func/sub-S02_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S03/func/sub-S03_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S04/func/sub-S04_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S05/func/sub-S05_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S06/func/sub-S06_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S07/func/sub-S07_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S08/func/sub-S08_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S09/func/sub-S09_task-localizer_bold.nii.gz'>,
 <BIDSImageFile filename='/Users/lukechang/Dropbox/Dartbrains/Data/localizer/sub-
S10/func/sub-S10_task-localizer_bold.nii.gz'>]
```

Notice that there are nifti and event files. We can get the filename for the first particant's functional run

```
f = layout.get(task='localizer')[0].filename
f
```

```
'sub-S01_task-localizer_desc-carpetplot_bold.svg'
```

If you want a summary of all the files in your BIDSLayout, but don't want to have to iterate BIDSFile objects and extract their entities, you can get a nice bird's-eye view of your dataset using the `to_df()` method.

```
layout.to_df()
```

entity	path	datatype	extension	subject	su
0	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	json	NaN	descript
1	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	json	NaN	participa
2	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	tsv	NaN	participa
3	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	tsv	NaN	behavio
4	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	tsv	NaN	subj
...	...	...	...	...	
284	/Users/lukechang/Dropbox/Dartbrains/Data/local...	anat	nii.gz	S94	T
285	/Users/lukechang/Dropbox/Dartbrains/Data/local...	func	nii.gz	S94	b
286	/Users/lukechang/Dropbox/Dartbrains/Data/local...	func	tsv	S94	eve
287	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	json	NaN	b
288	/Users/lukechang/Dropbox/Dartbrains/Data/local...	NaN	NaN	NaN	N

289 rows × 6 columns

Loading Data with Nibabel

Neuroimaging data is often stored in the format of nifti files `.nii` which can also be compressed using gzip `.nii.gz`. These files store both 3D and 4D data and also contain structured metadata in the image **header**.

There is an very nice tool to access nifti data stored on your file system in python called nibabel. If you don't already have nibabel installed on your computer it is easy via `pip`. First, tell the jupyter cell that you would like to access the unix system outside of the notebook and then install nibabel using pip `!pip install nibabel`. You only need to run this once (unless you would like to update the version).

nibabel objects can be initialized by simply pointing to a nifti file even if it is compressed through gzip. First, we will import the nibabel module as `nib` (short and sweet so that we don't have to type so much when using the tool). I'm also including a path to where the data file is located so that I don't have to constantly type this. It is easy to change this on your own computer.

We will be loading an anatomical image from subject S01 from the localizer dataset. See this paper for more information about this dataset.

We will use pybids to grab subject S01's T1 image.

```
import nibabel as nib

data = nib.load(layout.get(subject='S01', scope='derivatives', suffix='T1w',
return_type='file', extension='nii.gz')[1])
```

If we want to get more help on how to work with the nibabel data object we can either consult the documentation or add a `?`.

```
data?
```

The imaging data is stored in either a 3D or 4D numpy array. Just like numpy, it is easy to get the dimensions of the data using `shape`.

```
data.shape
```

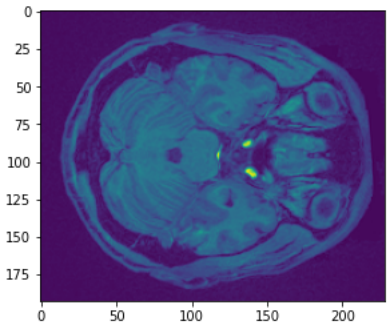
```
(193, 229, 193)
```

Looks like there are 3 dimensions (x,y,z) that is the number of voxels in each dimension. If we know the voxel size, we could convert this into millimeters.

We can also directly access the data and plot a single slice using standard matplotlib functions.

```
%matplotlib inline

import matplotlib.pyplot as plt

plt.imshow(data.get_fdata()[:,:,50])
```

```
<matplotlib.image.AxesImage at 0x7fadc1706e50>
```



Try slicing different dimensions (x,y,z) yourself to get a feel for how the data is represented in this anatomical image.

We can also access data from the image header. Let's assign the header of an image to a variable and print it to view it's contents.

```
header = data.header
print(header)
```

```
<class 'nibabel.nifti1.Nifti1Header'> object, endian='<'
sizeof_hdr      : 348
data_type       : b''
db_name         : b''
extents         : 0
session_error   : 0
regular         : b'r'
dim_info        : 0
dim             : [  3 193 229 193   1   1   1   1]
intent_p1       : 0.0
intent_p2       : 0.0
intent_p3       : 0.0
intent_code     : none
datatype        : float32
bitpix          : 32
slice_start     : 0
pixdim          : [1. 1. 1. 1. 0. 0. 0. 0.]
vox_offset      : 0.0
scl_slope       : nan
scl_inter       : nan
slice_end       : 0
slice_code      : unknown
xyzt_units      : 2
cal_max         : 0.0
cal_min         : 0.0
slice_duration  : 0.0
toffset         : 0.0
glmax           : 0
glmin           : 0
descrip         : b'xform matrices modified by FixHeaderApplyTransforms (niworkflows
v1.1.12).'
aux_file        : b''
qform_code      : mni
sform_code      : mni
quatern_b       : 0.0
quatern_c       : 0.0
quatern_d       : 0.0
qoffset_x       : -96.0
qoffset_y       : -132.0
qoffset_z       : -78.0
srow_x          : [  1.   0.   0. -96.]
srow_y          : [   0.    1.    0. -132.]
srow_z          : [  0.   0.   1. -78.]
intent_name     : b''
magic           : b'n+1'
```

Some of the important information in the header is information about the orientation of the image in space. This can be represented as the affine matrix, which can be used to transform images between different spaces.

```
data.affine
```

```
array([[   1.,    0.,    0.,  -96.],
       [   0.,    1.,    0., -132.],
       [   0.,    0.,    1.,  -78.],
       [   0.,    0.,    0.,    1.]])
```

We will dive deeper into affine transformations in the preprocessing tutorial.

Plotting Data with Nilearn

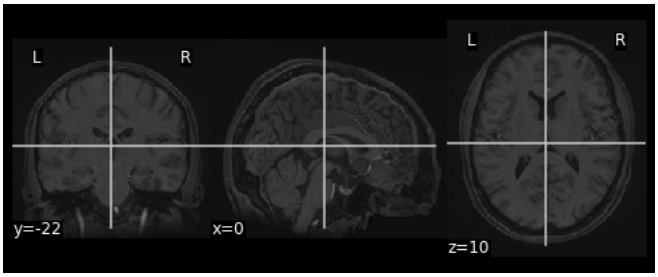
There are many useful tools from the nilearn library to help manipulate and visualize neuroimaging data. See their documentation for an example.

In this section, we will explore a few of their different plotting functions, which can work directly with nibabel instances.

```
%matplotlib inline

from nilearn.plotting import view_img, plot_glass_brain, plot_anat, plot_epi
```

```
plot_anat(data)
```

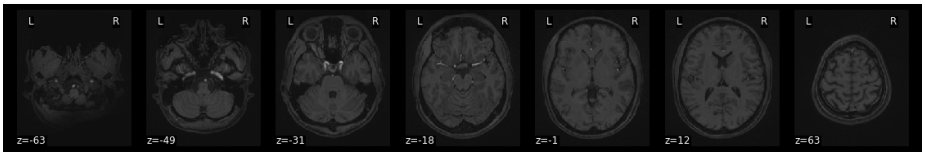
```
<nilearn.plotting.displays.OrthoSlicer at 0x7fadd13c4c50>
```



Nilearn plotting functions are very flexible and allow us to easily customize our plots

```
plot_anat(data, draw_cross=False, display_mode='z')
```

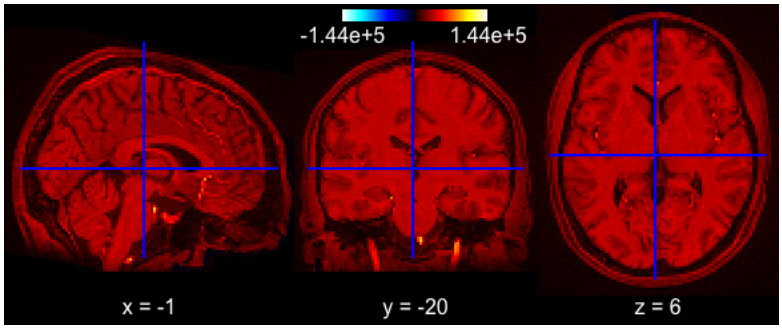
```
<nilearn.plotting.displays.ZSlicer at 0x7fadc157fd50>
```



try to get more information how to use the function with `?` and try to add different commands to change the plot.

nilearn also has a neat interactive viewer called `view_img` for examining images directly in the notebook.

```
view_img(data)
```

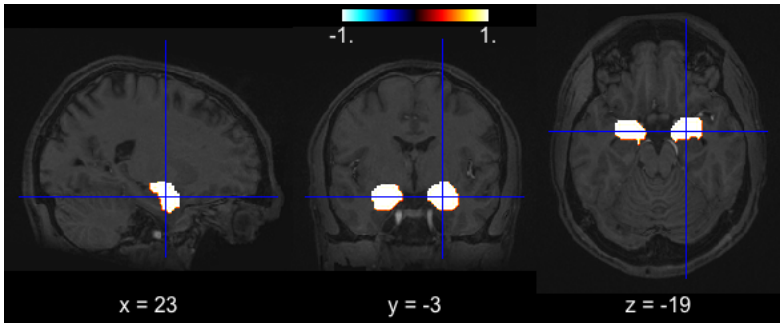


The `view_img` function is particularly useful for overlaying statistical maps over an anatomical image so that we can interactively examine where the results are located.

As an example, let's load a mask of the amygdala and try to find where it is located.

```
amygdala_mask = nib.load(os.path.join(data_dir, '../..', 'masks',
'FSL_BAmyg_thr0.nii.gz'))
view_img(amygdala_mask, data)
```

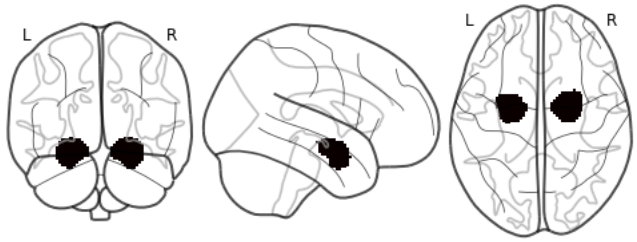
```
/Users/lukechang/anaconda3/lib/python3.7/site-
packages/nilearn/image/resampling.py:513: UserWarning: Casting data from int32 to
float32
  warnings.warn("Casting data from %s to %s" % (data.dtype.name, aux))
```



We can also plot a glass brain which allows us to see through the brain from different slice orientations. In this example, we will plot the binary amygdala mask.

```
plot_glass_brain(amygdala_mask)
```

```
<nilearn.plotting.displays.OrthoProjector at 0x7f92415be7d0>
```



Manipulating Data with Nltools

Ok, we've now learned how to use nibabel to load imaging data and nilearn to plot it.

Next we are going to learn how to use the `nltools` package that tries to make loading, plotting, and manipulating data easier. It uses many functions from nibabe, nilearn, and other python libraries. The bulk of the nltools toolbox is built around the `Brain_Data()` class. The concept behind the class is to have a similar feel to a pandas dataframe, which means that it should feel intuitive to manipulate the data.

The `Brain_Data()` class has several attributes that may be helpful to know about. First, it stores imaging data in `.data` as a vectorized features by observations matrix. Each image is an observation and each voxel is a feature. Space is flattened using `nifti_masker` from nilearn. This object is also stored as an attribute in `.nifti_masker` to allow transformations from 2D to 3D/4D matrices. In addition, a brain_mask is stored in `.mask`. Finally, there are attributes to store either class labels for prediction/classification analyses in `.Y` and design matrices in `.X`. These are both expected to be pandas `DataFrames`.

We will give a quick overview of basic Brain_Data operations, but we encourage you to see our documentation for more details.

Brain_Data basics

To get a feel for `Brain_Data`, let's load an example anatomical overlay image that comes packaged with the toolbox.

```
from nltools.data import Brain_Data
from nltools.utils import get_anatomical

anat = Brain_Data(get_anatomical())
anat
```

```
nltools.data.brain_data.Brain_Data(data=(238955,), Y=0, X=(0, 0),
mask=MNI152_T1_2mm_brain_mask.nii.gz, output_file=[])
```

To view the attributes of `Brain_Data` use the `vars()` function.

```
print(vars(anat))
```

```
{'mask': <nibabel.nifti1.Nifti1Image object at 0x7fad95829a50>, 'nifti_masker':
NiftiMasker(detrend=False, dtype=None, high_pass=None, low_pass=None,
            mask_args=None,
            mask_img=<nibabel.nifti1.Nifti1Image object at 0x7fad95829a50>,
            mask_strategy='background', memory=Memory(cachedir=None),
            memory_level=1, reports=True, sample_mask=None, sessions=None,
            smoothing_fwhm=None, standardize=False, t_r=None,
            target_affine=None, target_shape=None, verbose=0), 'data': array([1875.,
2127., 2182., ..., 5170., 5180., 2836.], dtype=float32), 'Y': Empty DataFrame
Columns: []
Index: [], 'X': Empty DataFrame
Columns: []
Index: [], 'file_name': []}
```

`Brain_Data` has many methods to help manipulate, plot, and analyze imaging data. We can use the `dir()` function to get a quick list of all of the available methods that can be used on this class.

To learn more about how to use these tools either use the `?` function, or look up the function in the api documentation.

```
print(dir(anat))
```

```
['X', 'Y', '__add__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__module__', '__mul__', '__ne__', '__new__', '__radd__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmul__', '__rsub__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__weakref__', 'aggregate',
'align', 'append', 'apply_mask', 'astype', 'bootstrap', 'copy', 'data', 'decompose',
'detrend', 'distance', 'dtype', 'empty', 'extract_roi', 'file_name', 'filter',
'find_spikes', 'groupby', 'icc', 'iplot', 'isempty', 'mask', 'mean', 'median',
'multivariate_similarity', 'nifti_masker', 'plot', 'predict', 'predict_multi',
'r_to_z', 'randomise', 'regions', 'regress', 'scale', 'shape', 'similarity',
'smooth', 'standardize', 'std', 'sum', 'threshold', 'to_nifti', 'transform_pairwise',
'ttest', 'upload_neurovault', 'write']
```

Ok, now let's load a single subject's functional data from the localizer dataset. We will load one that has already been preprocessed with fmriprep and is stored in the derivatives folder.

Loading data can be a little bit slow especially if the data need to be resampled to the template, which is set at $2mm^3$ by default. However, once it's loaded into the workspace it should be relatively fast to work with it.

```
data = Brain_Data(layout.get(target='subject', scope='derivatives', suffix='bold',
extension='nii.gz', return_type='file')[0])
```

Here are a few quick basic data operations.

Find number of images in Brain_Data() instance

```
print(len(data))
```

```
128
```

Find the dimensions of the data (images x voxels)

```
print(data.shape())
```

```
(128, 238955)
```

We can use any type of indexing to slice the data such as integers, lists of integers, slices, or boolean vectors.

```
import numpy as np

print(data[5].shape())

print(data[[1,6,2]].shape())

print(data[0:10].shape())

index = np.zeros(len(data), dtype=bool)
index[[1,5,9, 16, 20, 22]] = True

print(data[index].shape())
```

```
(238955,)
(3, 238955)
(10, 238955)
(6, 238955)
```

Simple Arithmetic Operations

Calculate the mean for every voxel over images

```
data.mean()
```

```
nltools.data.brain_data.Brain_Data(data=(238955,), Y=0, X=(0, 0),
mask=MNI152_T1_2mm_brain_mask.nii.gz, output_file=[])
```

Calculate the standard deviation for every voxel over images

```
data.std()
```

```
nltools.data.brain_data.Brain_Data(data=(238955,), Y=0, X=(0, 0),
mask=MNI152_T1_2mm_brain_mask.nii.gz, output_file=[])
```

Methods can be chained. Here we get the shape of the mean.

```
print(data.mean().shape())
```

```
(238955,)
```

Brain_Data instances can be added and subtracted

```
new = data[1]+data[2]
```

Brain_Data instances can be manipulated with basic arithmetic operations.

Here we add 10 to every voxel and scale by 2

```
data2 = (data + 10) * 2
```

Brain_Data instances can be copied

```
new = data.copy()
```

Brain_Data instances can be easily converted to nibabel instances, which store the data in a 3D/4D matrix. This is useful for interfacing with other python toolboxes such as nilearn

```
data.to_nifti()
```

```
<nibabel.nifti1.Nifti1Image at 0x7fad966b7d10>
```

Brain_Data instances can be concatenated using the append method

```
new = new.append(data[4])
```

Lists of `Brain_Data` instances can also be concatenated by recasting as a `Brain_Data` object.

```
print(type([x for x in data[:4]]))

type(Brain_Data([x for x in data[:4]]))
```

```
<class 'list'>
```

```
nltools.data.brain_data.Brain_Data
```

Any Brain_Data object can be written out to a nifti file.

```
data.write('Tmp_Data.nii.gz')
```

Images within a Brain_Data() instance are iterable. Here we use a list comprehension to calculate the overall mean across all voxels within an image.

```
[x.mean() for x in data]
```

```
[3631.2645411383587,
 3637.836965559738,
 3634.8721272399216,
 3629.901022394308,
 3625.903680905571,
 3629.8617967834693,
 3647.0626827990523,
 3656.532783431514,
 3652.9664477289007,
 3656.854690370624,
 3651.158568986219,
 3646.0627760781335,
 3647.096514090842,
 3650.3529496858014,
 3647.3099736730073,
 3647.541164716155,
 3653.0196194154546,
 3653.9954711802293,
 3648.761361342424,
 3643.4910051915754,
 3644.4916183059545,
 3643.806287863885,
 3632.513377052877,
 3634.0742188451927,
 3632.823087421774,
 3636.538413516169,
 3635.2725020511252,
 3641.7997040537466,
 3641.7916592727656,
 3634.091262543454,
 3643.245243803917,
 3658.5574913389905,
 3649.2724227942545,
 3641.1560643044045,
 3643.306476451434,
 3637.0167340942126,
 3637.5381102659767,
 3644.4683410479392,
 3639.8860350581976,
 3630.9967110874504,
 3623.7886672981303,
 3627.072535710563,
 3624.862147866512,
 3622.6118047366986,
 3634.9950195308434,
 3627.3577482514256,
 3628.0705544219018,
 3621.129485290891,
 3616.061808421585,
 3608.6081438384495,
 3619.78010303786,
 3625.2043623406903,
 3630.7597935028207,
 3628.074661527296,
 3630.138505445051,
 3624.1916798686207,
 3620.2774727712167,
 3619.0700248246812,
 3624.4436554897025,
 3625.009898536367,
 3622.018871417351,
 3629.2243599621006,
 3629.480077460646,
 3625.544210353266,
 3621.4695822776903,
 3617.3724419581886,
 3615.911084856664,
 3614.292718966583,
 3616.767158878271,
 3621.5679265347426,
 3617.094424744487,
 3609.954978219352,
 3612.64425239096,
 3629.0631390560125,
 3628.8013229265916,
 3621.5916670872352,
 3612.4806693745804,
 3613.664459683712,
 3621.6783653848643,
 3621.143953281178,
 3618.5757894664835,
 3610.795271247279,
 3613.4845495891113,
 3607.408626131088,
 3613.440184819298,
 3608.650315768319,
 3604.885826645298,
```

```
3601.629542348782,
3600.8264436489476,
3600.6285381425578,
3609.776906766102,
3618.2023236145965,
3615.7040390888783,
3612.2955410039767,
3606.200637253472,
3623.39621843859,
3627.132987563558,
3611.296930448837,
3594.7923511331287,
3580.0196095574156,
3579.4805425330965,
3583.0559348949373,
3592.5569828498697,
3604.371337719925,
3606.9999113264666,
3620.9576329138504,
3617.235063465483,
3612.5929641090283,
3605.798535577046,
3597.116219752562,
3588.5229574072764,
3587.5091257654085,
3598.2696343420794,
3602.2806983812598,
3601.700440752411,
3610.3855186675264,
3610.527882720576,
3604.287286181301,
3591.7958436327026,
3591.129841385018,
3598.8536387360964,
3611.565702027962,
3610.1719483427555,
3618.0720138472166,
3612.4640399870646,
3598.581193499466,
3594.3173923249437,
3595.754862354543]
```

Though, we could also do this with the `mean` method by setting `axis=1`.

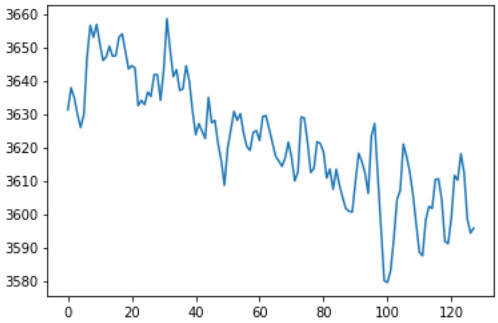
```
data.mean(axis=1)
```

```
array([3631.26454114, 3637.83696556, 3634.87212724, 3629.90102239,
       3625.90368091, 3629.86179678, 3647.0626828 , 3656.53278343,
       3652.96644773, 3656.85469037, 3651.15856899, 3646.06277608,
       3647.09651409, 3650.35294969, 3647.30997367, 3647.54116472,
       3653.01961942, 3653.99547118, 3648.76136134, 3643.49100519,
       3644.49161831, 3643.80628786, 3632.51337705, 3634.07421885,
       3632.82308742, 3636.53841352, 3635.27250205, 3641.79970405,
       3641.79165927, 3634.09126254, 3643.2452438 , 3658.55749134,
       3649.27242279, 3641.1560643 , 3643.30647645, 3637.01673409,
       3637.53811027, 3644.46834105, 3639.88603506, 3630.99671109,
       3623.7886673 , 3627.07253571, 3624.86214787, 3622.61180474,
       3634.99501953, 3627.35774825, 3628.07055442, 3621.12948529,
       3616.06180842, 3608.60814384, 3619.78010304, 3625.20436234,
       3630.7597935 , 3628.07466153, 3630.13850545, 3624.19167987,
       3620.27747277, 3619.07002482, 3624.44365549, 3625.00989854,
       3622.01887142, 3629.22435996, 3629.48007746, 3625.54421035,
       3621.46958228, 3617.37244196, 3615.91108486, 3614.29271897,
       3616.76715888, 3621.56792653, 3617.09442474, 3609.95497822,
       3612.64425239, 3629.06313906, 3628.80132293, 3621.59166709,
       3612.48066937, 3613.66445968, 3621.67836538, 3621.14395328,
       3618.57578947, 3610.79527125, 3613.48454959, 3607.40862613,
       3613.44018482, 3608.65031577, 3604.88582665, 3601.62954235,
       3600.82644365, 3600.62853814, 3609.77690677, 3618.20232361,
       3615.70403909, 3612.295541  , 3606.20063725, 3623.39621844,
       3627.13298756, 3611.29693045, 3594.79235113, 3580.01960956,
       3579.48054253, 3583.05593489, 3592.55698285, 3604.37133772,
       3606.99991133, 3620.95763291, 3617.23506347, 3612.59296411,
       3605.79853558, 3597.11621975, 3588.52295741, 3587.50912577,
       3598.26963434, 3602.28069838, 3601.70044075, 3610.38551867,
       3610.52788272, 3604.28728618, 3591.79584363, 3591.12984139,
       3598.85363874, 3611.56570203, 3610.17194834, 3618.07201385,
       3612.46403999, 3598.5811935 , 3594.31739232, 3595.75486235])
```

Let's plot the mean to see how the global signal changes over time.

```
plt.plot(data.mean(axis=1))
```

```
[<matplotlib.lines.Line2D at 0x7fad966a6590>]
```



Notice the slow linear drift over time, where the global signal intensity gradually decreases. We will learn how to remove this with a high pass filter in future tutorials.

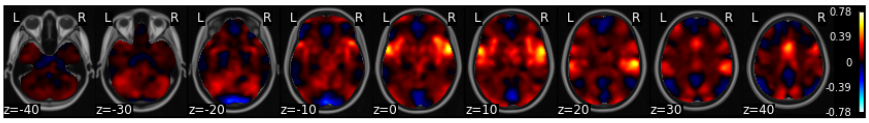
Plotting

There are multiple ways to plot your data.

For a very quick plot, you can return a montage of axial slices with the `.plot()` method. As an example, we will plot the mean of each voxel over time.

```
f = data.mean().plot()
```

```
threshold is ignored for simple axial plots
```



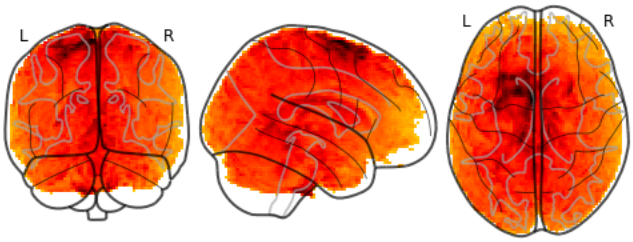
There is an interactive `.iplot()` method based on nilearn `view_img`.

```
data.mean().iplot()
```

Brain_Data() instances can be converted to a nibabel instance and plotted using any nilearn plot method such as glass brain.

```
plot_glass_brain(data.mean().to_nifti())
```

```
<nilearn.plotting.displays.OrthoProjector at 0x7fad07c45a10>
```



Ok, that's the basics. `Brain_Data` can do much more!

Check out some of our tutorials for more detailed examples.

We'll be using this tool throughout the course.

Exercises

For homework, let's practice our skills in working with data.

Exercise 1

A few subjects have already been preprocessed with fMRI prep.

Use pybids to figure out which subjects have been preprocessed.

Exercise 2

One question we are often interested in is where in the brain do we have an adequate signal to noise ratio (SNR). There are many different metrics, here we will use temporal SNR, which the voxel mean over time divided by it's standard deviation.

$$\text{tSNR} = \frac{\text{mean}(\text{voxel}_i)}{\text{std}(\text{voxel}_i)}$$

In Exercise 2, calculate the SNR for S01 and plot this so we can figure which regions have high and low SNR.

Exercise 3

We are often interested in identifying outliers in our data. In this exercise, find any image from 'S01' that exceeds a zscore of 2 and plot each one.

Signal Processing Basics

Written by Luke Chang

In this lab, we will cover the basics of convolution, sine waves, and fourier transforms. This lab is largely based on exercises from Mike X Cohen's excellent book, Analying Neural Data Analysis: Theory and Practice. If you are interested in learning in more detail about the basics of EEG and time-series analyses I highly recommend his accessible introduction. I also encourage you to watch his accompanying freely available *lecturelets* to learn more about each topic introduced in this notebook.

Time Domain

First we will work on signals in the time domain. This requires measuring a signal at a constant interval over time. The frequency with which we measure a signal is referred to as the sampling frequency. The units of this are typically described in Hz - or the number of cycles per second. It is critical that the sampling frequency is consistent over the entire measurement of the time series.

Dot Product

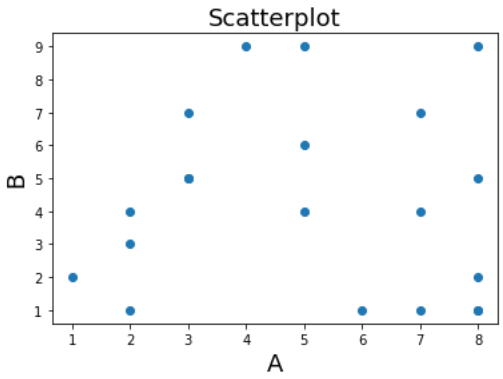
To understand convolution, we first need to familiarize ourselves with the dot product. The dot product is simply the sum of the elements of a vector weighted by the elements of another vector. This method is commonly used in signal processing, and also in statistics as a measure of similarity between two vectors. Finally, there is also a geometric inrepretation which is a mapping between vectors (i.e., the product of the magnitudes of the two vectors scaled by the cosine of the angle between them). For a more in depth overview of the dot product and its relation to convolution, you can watch this optional video.

$$dotproduct_{ab} = \sum_{i=1}^{n} a_i b_i$$

Let's create some vectors of random numbers and see how the dot product works. First, the two vectos need to be of the same length.

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

a = np.random.randint(1,10,20)
b = np.random.randint(1,10,20)

plt.scatter(a,b)
plt.ylabel('B', fontsize=18)
plt.xlabel('A', fontsize=18)
plt.title('Scatterplot', fontsize=18)

print('Dot Product: %s' % np.dot(a,b))
```

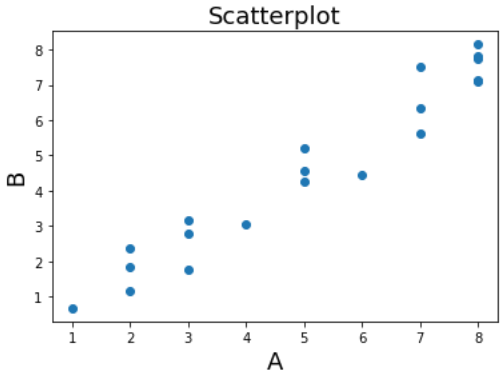
```
Dot Product: 434
```



what happens when we make the two variables more similar? In the next example we add gaussian noise on top of one of the vectors. What happens to the dot product?

```
b = a + np.random.randn(20)
plt.scatter(a,b)
plt.ylabel('B', fontsize=18)
plt.xlabel('A', fontsize=18)
plt.title('Scatterplot', fontsize=18)

print(f'Dot Product: {np.dot(a,b)}')
```

```
Dot Product: 583.1197152049825
```



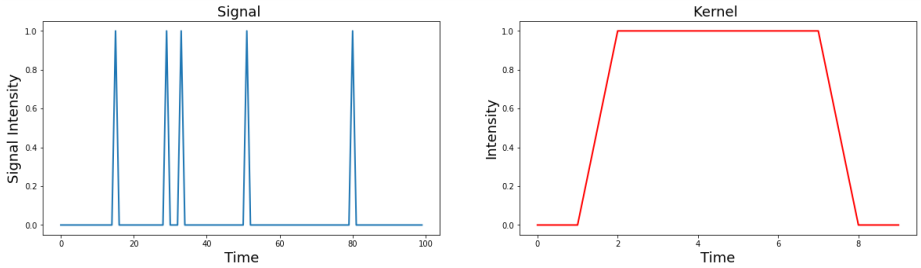
Convolution

Convolution in the time domain is an extension of the dot product in which the dot product is computed iteratively over time. One way to think about it is that one signal weights each time point of the other signal and then slides forward over time. Let's call the timeseries variable *signal* and the other vector the *kernel*. Importantly, for our purposes, the kernel will almost always be smaller than the signal, otherwise we would only have one scalar value afterwards.

To gain an intuition of how convolution works, let's play with some data. First, let's create a time series of spikes. Then let's convolve this signal with a boxcar kernel.

```
n_samples = 100

signal = np.zeros(n_samples)
signal[np.random.randint(0 ,n_samples, 5)] = 1

kernel = np.zeros(10)
kernel[2:8] = 1

f,a = plt.subplots(ncols=2, figsize=(20,5))
a[0].plot(signal, linewidth=2)
a[0].set_xlabel('Time', fontsize=18)
a[0].set_ylabel('Signal Intensity', fontsize=18)
a[0].set_title('Signal ', fontsize=18)
a[1].plot(kernel, linewidth=2, color='red')
a[1].set_xlabel('Time', fontsize=18)
a[1].set_ylabel('Intensity', fontsize=18)
a[1].set_title('Kernel ', fontsize=18)
```

```
Text(0.5, 1.0, 'Kernel ')
```



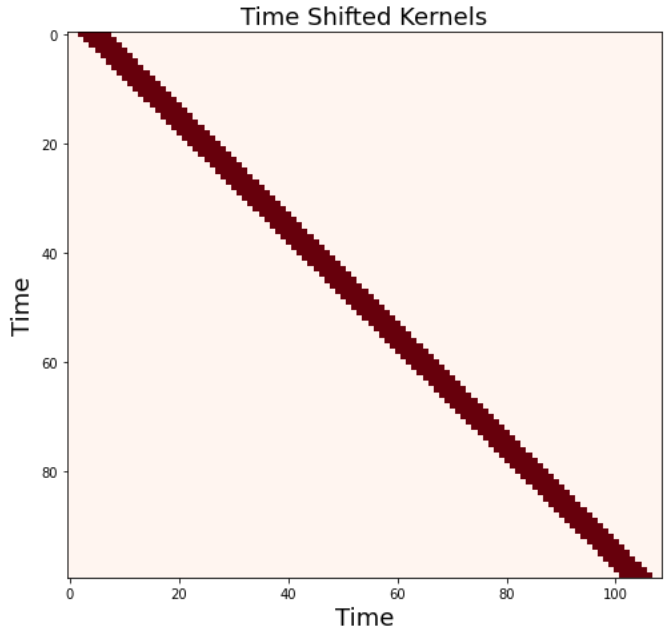
Notice how the kernel is only 10 samples long and the boxcar width is about 6 seconds, while the signal is 100 samples long with 5 single pulses.

Now let's convolve the signal with the kernel by taking the dot product of the kernel with each time point of the signal. This can be illustrated by creating a matrix of the kernel shifted each time point of the signal.

We will illustrate using a heatmap, where the change in the color reflects the intensity, that this is simply moving the boxcar kernel, which is 6 seconds in duration forward in time for each sample.

```
shifted_kernel = np.zeros((n_samples, n_samples+len(kernel) - 1))
for i in range(n_samples):
    shifted_kernel[i, i:i+len(kernel)] = kernel

plt.figure(figsize=(8, 8))
plt.imshow(shifted_kernel, cmap='Reds')
plt.xlabel('Time', fontsize=18)
plt.ylabel('Time', fontsize=18)
plt.title('Time Shifted Kernels', fontsize=18)
```

```
Text(0.5, 1.0, 'Time Shifted Kernels')
```

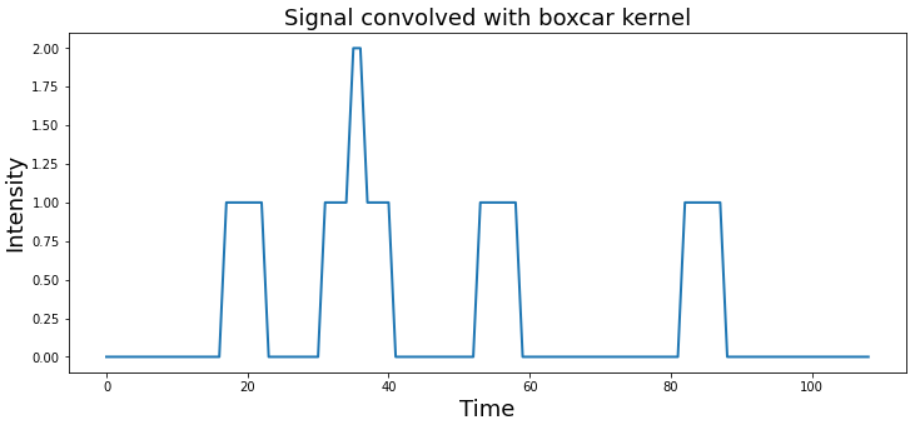


Now, let's take the dot product of the signal with this matrix.

To refresh your memory from basic linear algebra. Matrix multiplication consists of taking the dot product of the signal vector with each row of this expanded kernel matrix.

```
convolved_signal = np.dot(signal, shifted_kernel)

plt.figure(figsize=(12, 5))
plt.plot(convolved_signal, linewidth=2)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.title('Signal convolved with boxcar kernel', fontsize=18)
```

```
Text(0.5, 1.0, 'Signal convolved with boxcar kernel')
```



You can see that after convolution, each spike has now become the shape of the kernel. Spikes that were closer in time, compound if the boxes overlap.

Notice also how the shape of the final signal is the length of the combined signal and kernel minus one.

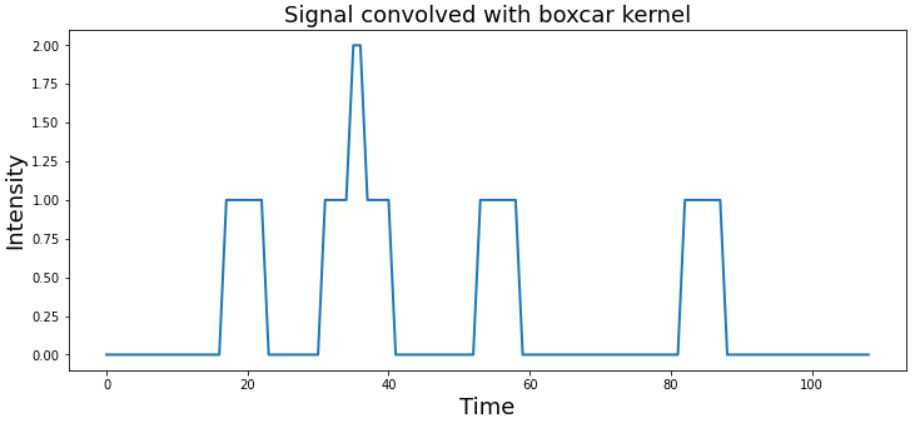
```
print(f"Signal Length: {len(signal)}")
print(f"Kernel Length: {len(kernel)}")
print(f"Convolved Signal Length: {len(convolved_signal)}")
```

```
Signal Length: 100
Kernel Length: 10
Convolved Signal Length: 109
```

this process of iteratively taking the dot product of the kernel with each timepoint of the signal and summing all of the values can be performed by using the convolution function from numpy `np.convolve`

```
plt.figure(figsize=(12, 5))
plt.plot(np.convolve(signal, kernel), linewidth=2)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.title('Signal convolved with boxcar kernel', fontsize=18)
```

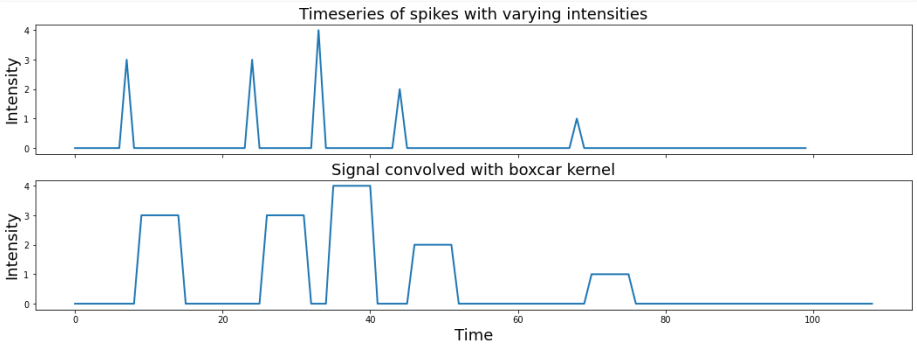
```
Text(0.5, 1.0, 'Signal convolved with boxcar kernel')
```



What happens if the spikes have different intensities, reflected by different heights?

```
signal = np.zeros(n_samples)
signal[np.random.randint(0,n_samples,5)] = np.random.randint(1,5,5)

f,a = plt.subplots(nrows=2, figsize=(18,6), sharex=True)
a[0].plot(signal, linewidth=2)
a[0].set_ylabel('Intensity', fontsize=18)
a[0].set_title('Timeseries of spikes with varying intensities', fontsize=18)
a[1].plot(np.convolve(signal, kernel), linewidth=2)
a[1].set_ylabel('Intensity', fontsize=18)
a[1].set_title('Signal convolved with boxcar kernel', fontsize=18)
a[1].set_xlabel('Time', fontsize=18)
```

```
Text(0.5, 0, 'Time')
```



Now what happens if we switch out the boxcar kernel for something with a more interesting shape, say a hemodynamic response function?

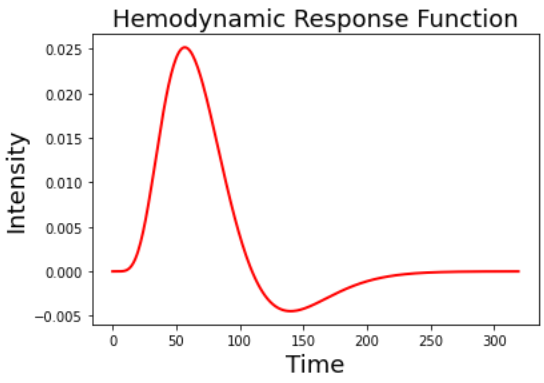
Here we will use a double gamma hemodynamic function (HRF) developed by Gary Glover.

Note: If you haven't install nltools yet run `!pip install nltools`. You may need to restart your jupyter kernel as well.

```
from nltools.external import glover_hrf

tr = 2
hrf = glover_hrf(tr, oversampling=20)
plt.plot(hrf, linewidth=2, color='red')
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.title('Hemodynamic Response Function', fontsize=18)
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-
packages/sklearn/utils/deprecation.py:144: FutureWarning: The
sklearn.linear_model.base module is  deprecated in version 0.22 and will be removed
in version 0.24. The corresponding classes / functions should instead be imported
from sklearn.linear_model. Anything that cannot be imported from sklearn.linear_model
is now part of the private API.
  warnings.warn(message, FutureWarning)
```

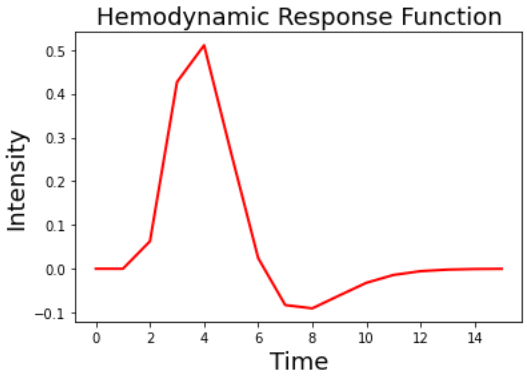
```
Text(0.5, 1.0, 'Hemodynamic Response Function')
```



For this example, we oversampled the function to make it more smooth. In practice we will want to make sure that the kernel is the correct shape given our sampling resolution. Be sure to se the oversampling to 1. Notice how the function looks more jagged now?

```
hrf = glover_hrf(tr, oversampling=1)
plt.plot(hrf, linewidth=2, color='red')
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.title('Hemodynamic Response Function', fontsize=18)
```

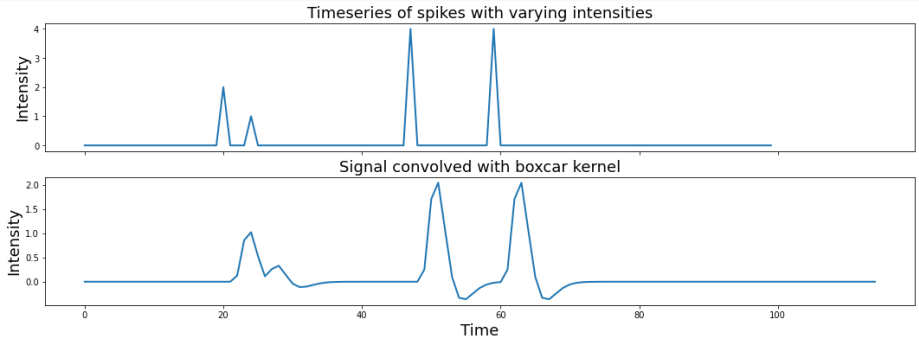
```
Text(0.5, 1.0, 'Hemodynamic Response Function')
```



Now let's try convolving our event pulses with this HRF kernel.

```
signal = np.zeros(n_samples)
signal[np.random.randint(0,n_samples,5)] = np.random.randint(1,5,5)

f,a = plt.subplots(nrows=2, figsize=(18,6), sharex=True)
a[0].plot(signal, linewidth=2)
a[1].plot(np.convolve(signal, hrf), linewidth=2)
a[0].set_ylabel('Intensity', fontsize=18)
a[0].set_title('Timeseries of spikes with varying intensities', fontsize=18)
a[1].set_ylabel('Intensity', fontsize=18)
a[1].set_xlabel('Time', fontsize=18)
a[1].set_title('Signal convolved with boxcar kernel', fontsize=18)
```

```
Text(0.5, 1.0, 'Signal convolved with boxcar kernel')
```



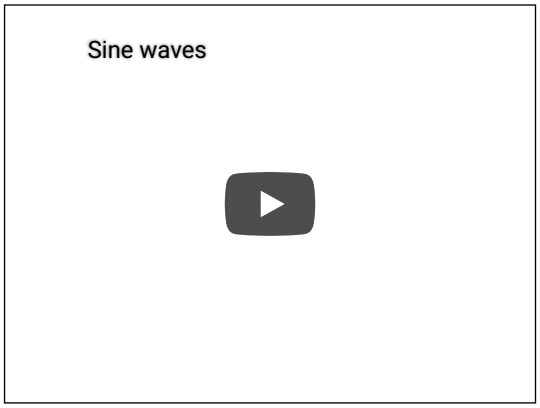
If you are interested in a more detailed overview of convolution in the time domain, I encourage you to watch this video by Mike X Cohen. For more details about convolution and the HRF function, see this overview using python examples.

Oscillations

Ok, now let's move on to studying time-varying signals that have the shape of oscillating waves.

Let's watch a short video by Mike X Cohen to get some more background on sine waves. Don't worry too much about the matlab code as we will work through similar Python examples in this notebook.

```
from IPython.display import YouTubeVideo

YouTubeVideo('9RvZXZ46FRQ')
```



Oscillations can be described mathematically as:

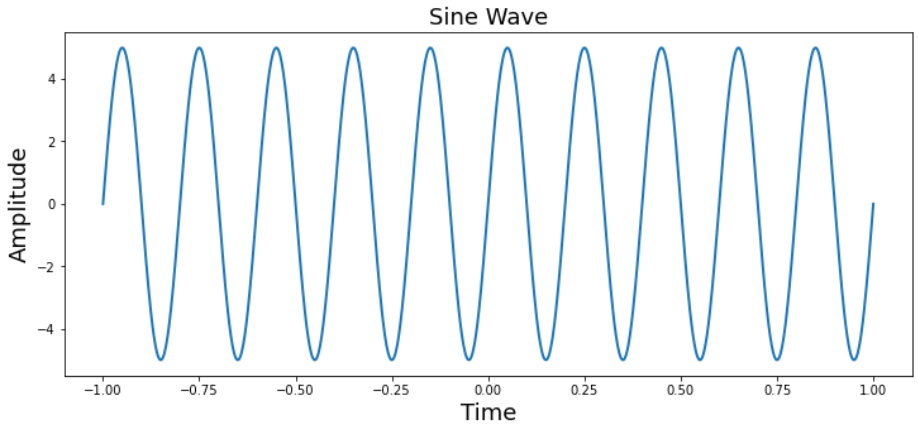
$A \sin(2\pi f t + \theta)$

where f is the frequency or the speed of the oscillation described in the number of cycles per second - Hz. Amplitude A refers to the height of the waves, which is half the distance of the peak to the trough. Finally, θ describes the phase angle offset, which is in radians.

Here we will plot a simple sine wave. Try playing with the different parameters (i.e., amplitude, frequency, & theta) to gain an intuition of how they each impact the shape of the wave.

```
from numpy import sin, pi, arange

sampling_freq = 500
time = arange(-1, 1 + 1/sampling_freq, 1/sampling_freq)
amplitude = 5
freq = 5
theta = 0

simulation = amplitude * sin(2 * pi * freq * time + theta)

plt.figure(figsize=(12, 5))
plt.plot(time, simulation, linewidth=2)
plt.title('Sine Wave', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.ylabel('Amplitude', fontsize=18)
```

```
Text(0, 0.5, 'Amplitude')
```



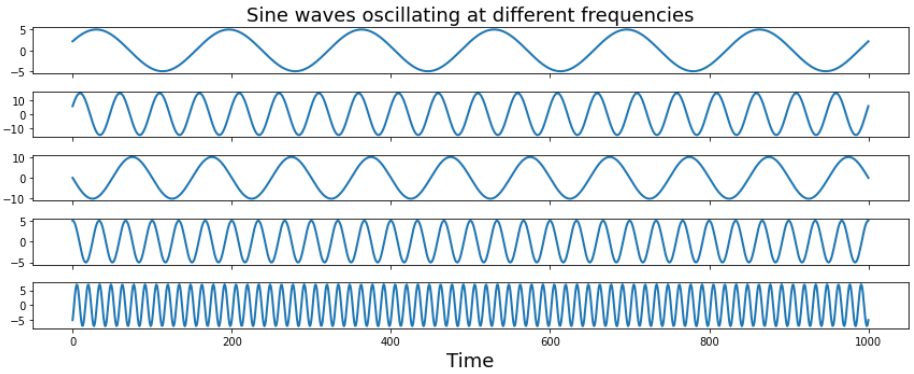
We can also see the impact of different parameters using interactive widgets. Here you can move the sliders to see the impact of varying the amplitude, frequency, and theta parameter on a sine wave. We also show the complex components of the sine wave in the right panel.

```
from ipywidgets import interact, FloatSlider
from numpy import sin, pi, arange, real, imag

def plot_oscillation(amplitude=5, frequency=5, theta=1):
    sampling_frequency=500
    time = arange(-1, 1 + 1/sampling_frequency, 1/sampling_frequency)
    simulation = amplitude * sin(2 * pi * frequency * time + theta)
    z = np.exp(1j*(2 * pi * frequency * time + theta))

    fig = plt.figure(figsize=(20, 4))
    gs = plt.GridSpec(1, 6, left=0.05, right=0.48, wspace=0.05)
    ax1 = fig.add_subplot(gs[0, :4])
    ax1.plot(time, simulation, linewidth=2)
    ax1.set_ylabel('Amplitude', fontsize=18)
    ax1.set_xlabel('Time', fontsize=18)
    ax2 = fig.add_subplot(gs[0, 5:], polar=True)
    ax2.plot(real(simulation), imag(simulation))
    plt.tight_layout()

interact(plot_oscillation, amplitude=FloatSlider(value=5, min=0, max=10, step=0.5),
         frequency=FloatSlider(value=5, min=0, max=10, step=0.5),
         theta=FloatSlider(value=0, min=-5, max=5, step=0.5))
```

```
<function __main__.plot_oscillation(amplitude=5, frequency=5, theta=1)>
```

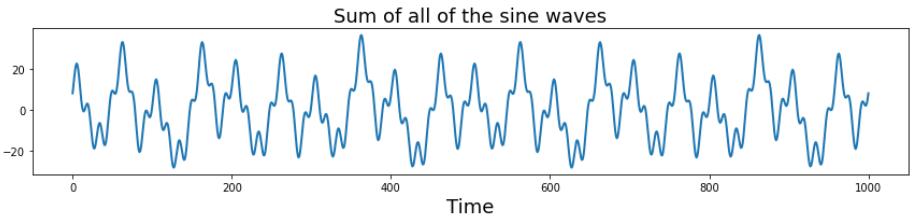
Next we will generate a simulation combining multiple sine waves oscillating at different frequencies.

```
sampling_freq = 500

freq = [3, 10, 5 ,15, 35]
amplitude = [5, 15, 10, 5, 7]
phases = pi*np.array([1/7, 1/8, 1, 1/2, -1/4])

time = arange(-1, 1 + 1/sampling_freq, 1/sampling_freq)

sine_waves = []
for i,f in enumerate(freq):
    sine_waves.append(amplitude[i] * sin(2*pi*f*time + phases[i]))
sine_waves = np.array(sine_waves)


f,a = plt.subplots(nrows=5, ncols=1, figsize=(12,5), sharex=True)
for i,x in enumerate(freq):
    a[i].plot(sine_waves[i,:], linewidth=2)
a[0].set_title("Sine waves oscillating at different frequencies", fontsize=18)
a[i].set_xlabel("Time", fontsize=18)
plt.tight_layout()
```



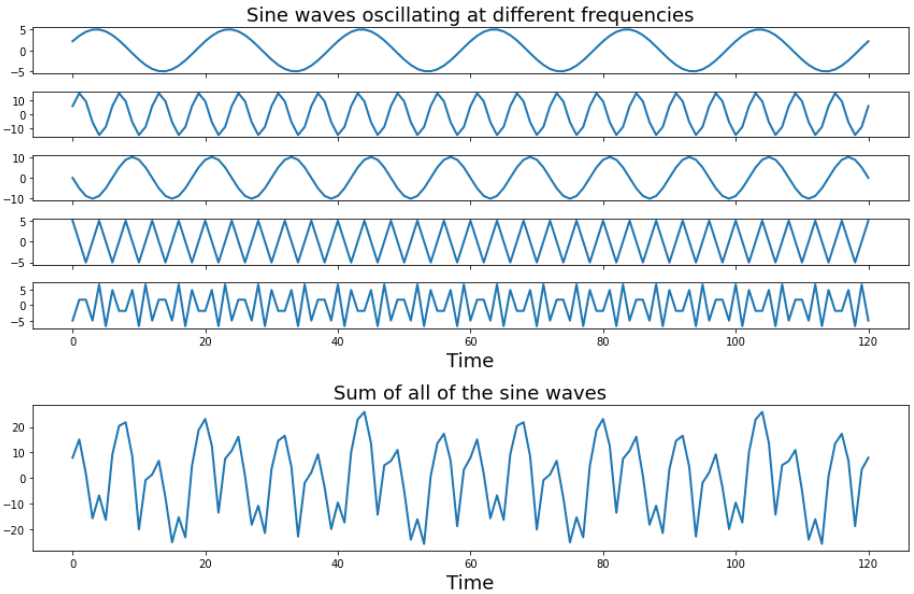
Let's add all of those signals together to get a more complex signal.

```
plt.figure(figsize=(12,3))
plt.plot(np.sum(sine_waves, axis=0), linewidth=2)
plt.xlabel('Time', fontsize=18)
plt.title("Sum of all of the sine waves", fontsize=18)
plt.xlabel("Time", fontsize=18)
plt.tight_layout()
```



What is the effect of changing the sampling frequency on our ability to measure these oscillations? Try dropping it to be very low (e.g., less than 70 hz.) Notice that signals will alias when the sampling frequency is below the nyquist frequency of a signal. To observe the oscillations, we need to be sampling at least two times for each oscillation cycle. This will result in a jagged view of the data, but we can still theoretically observe the frequency. Practically, higher sampling rates allow us to better observe the underlying signals.

```
sampling_freq = 60

freq = [3, 10, 5 ,15, 35]
amplitude = [5, 15, 10, 5, 7]
phases = pi*np.array([1/7, 1/8, 1, 1/2, -1/4])

time = arange(-1, 1 + 1/sampling_freq, 1/sampling_freq)

sine_waves = []
for i,f in enumerate(freq):
    sine_waves.append(amplitude[i] * sin(2*pi*f*time + phases[i]))
sine_waves = np.array(sine_waves)


f,a = plt.subplots(nrows=5, ncols=1, figsize=(12,5), sharex=True)
for i,x in enumerate(freq):
    a[i].plot(sine_waves[i,:], linewidth=2)
a[0].set_title("Sine waves oscillating at different frequencies", fontsize=18)
a[i].set_xlabel("Time", fontsize=18)
plt.tight_layout()


plt.figure(figsize=(12,3))
plt.plot(np.sum(sine_waves, axis=0), linewidth=2)
plt.title("Sum of all of the sine waves", fontsize=18)
plt.xlabel("Time", fontsize=18)
plt.tight_layout()
```

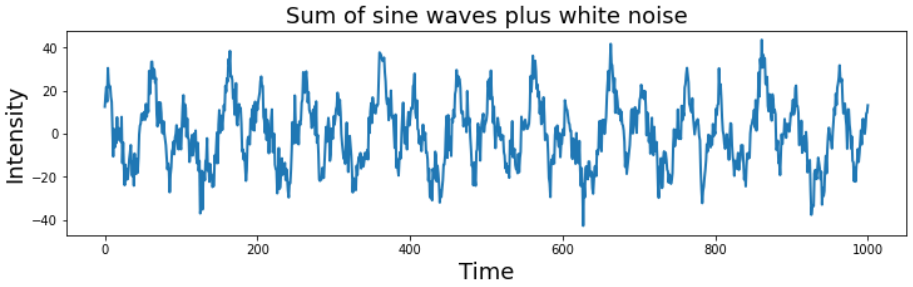


Notice the jagged lines for frequencies that are above the nyquist frequency? That's because we don't have enough samples to accurately see the oscillations.

Ok, let's increase the sampling frequency to remove the aliasing. We can add a little bit of gaussian (white) noise on top of this signal to make it even more realistic. Try varying the amount of noise by adjusting the scaling on the noise.

```
sampling_freq = 500

freq = [3, 10, 5 ,15, 35]
amplitude = [5, 15, 10, 5, 7]
phases = pi*np.array([1/7, 1/8, 1, 1/2, -1/4])

time = arange(-1, 1 + 1/sampling_freq, 1/sampling_freq)

sine_waves = []
for i,f in enumerate(freq):
    sine_waves.append(amplitude[i] * sin(2*pi*f*time + phases[i]))
sine_waves = np.array(sine_waves)


noise = 5 * np.random.randn(sine_waves.shape[1])
signal = np.sum(sine_waves,axis=0) + noise

plt.figure(figsize=(12,3))
plt.plot( signal, linewidth=2)
plt.title("Sum of sine waves plus white noise", fontsize=18)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
```

```
Text(0.5, 0, 'Time')
```



Time & Frequency Domains

We have seen above how to represent signals in the time domain. However, these signals can also be represented in the frequency domain.

Let's get started by watching a short video by Mike X Cohen to get an overview of how a signal can be represented in both of these different domains.

```
YouTubeVideo('fYtVHhk3xJ0')
```

Frequency Domain

In the previous example, we generated a complex signal composed of multiple sine waves oscillating at different frequencies. Typically in data analysis, we only observe the signal and are trying to uncover the generative processes that gave rise to the signal. In this section, we will introduce the frequency domain and how we can identify if there are any frequencies oscillating at a consistent frequency in our signal using the fourier transform. The fourier transform is essentially convolving different frequencies of sine waves with our data.

One important assumption to note is that the fourier transformations assume that your oscillatory signals are stationary, which means that the generative processes giving rise to the oscillations do not vary over time.

See this video for a more in depth discussion on stationarity. In practice, this assumption is rarely true. Often it can be useful to use other techniques such as wavelets to look at time x frequency representations. We will not be covering wavelets here, but see this series of videos for more information.

Discrete Time Fourier Transform

We will gain an intution of how the fourier transform works by building our own discrete time fourier transform.

Let's watch this short video about the fourier transform by Mike X Cohen. Don't worry too much about the details of the discussion on the matlab code as we will be exploring these concepts in python below.

```
YouTubeVideo('_htCsieA0_U')
```

The discrete Fourier transform of variable x at frequency f can be defined as:

$X_f = \sum_{k=0}^{n-1} x_k e^{-i2\pi f(k-1)n^{-1}}$ where n refers to the number of data points in vector x, and the capital letter X_f is the fourier coefficient of time series variable x at frequency f.

Essentially, we create a bank of complex sine waves at different frequencies that are linearly spaced. The zero frequency component reflects the mean offset over the entire signal and will simply be zero in our example.

Complex Sine Waves

You may have noticed that we are computing *complex* sine waves using the `np.exp` function instead of the `np.sin` function.

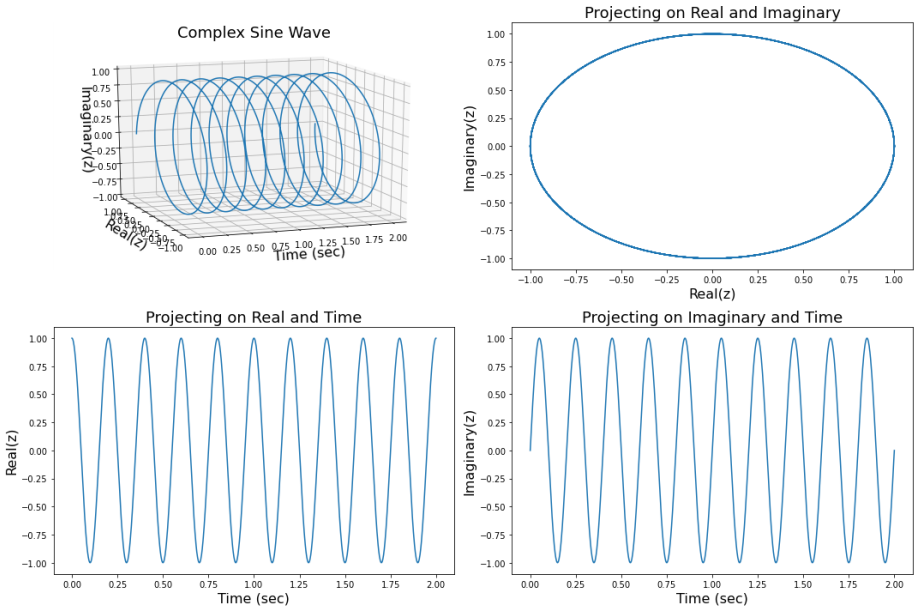
$$\text{complex sine wave} = e^{i(2\pi ft+\theta)}$$

We will not spend too much time on the details, but basically complex sine waves have three components: time, a real part of the sine wave, and the imaginary part of the sine wave, which are basically phase shifted by $\frac{\pi}{2}$. `1j` is how we can specify a complex number in python. We can extract the real components using `np.real` or the imaginary using `np.imag`.

We can visualize complex sine waves in three dimensions. For more information, watch this video. If you need a refresher on complex numbers, you may want to watch this video.

In this plot we show this complex signal in 3 dimensions and also project on two dimensional planes to show that the real and imaginary create a unit circle, and are phase offset by $\frac{\pi}{2}$ with respect to time.

```
from mpl_toolkits import mplot3d

frequency = 5
z =  np.exp(1j*(2 * pi * frequency * time + theta))

fig= plt.figure(figsize=(15, 10))
ax = fig.add_subplot(2, 2, 1, projection='3d')
ax.plot(np.arange(0, len(time))/sampling_freq, real(z), imag(z))
ax.set_xlabel('Time (sec)', fontsize=16)
ax.set_ylabel('Real(z)', fontsize=16)
ax.set_zlabel('Imaginary(z)', fontsize=16)
ax.set_title('Complex Sine Wave', fontsize=18)
ax.view_init(15, 250)

ax = fig.add_subplot(2, 2, 2)
ax.plot(real(z), imag(z))
ax.set_xlabel('Real(z)', fontsize=16)
ax.set_ylabel('Imaginary(z)', fontsize=16)
ax.set_title('Projecting on Real and Imaginary', fontsize=18)

ax = fig.add_subplot(2, 2, 3)
ax.plot(np.arange(0, len(time))/sampling_freq, real(z))
ax.set_xlabel('Time (sec)', fontsize=16)
ax.set_ylabel('Real(z)', fontsize=16)
ax.set_title('Projecting on Real and Time', fontsize=18)

ax = fig.add_subplot(2, 2, 4,)
ax.plot(np.arange(0, len(time))/sampling_freq, imag(z))
ax.set_xlabel('Time (sec)', fontsize=16)
ax.set_ylabel('Imaginary(z)', fontsize=16)
ax.set_title('Projecting on Imaginary and Time', fontsize=18)
plt.tight_layout()
```

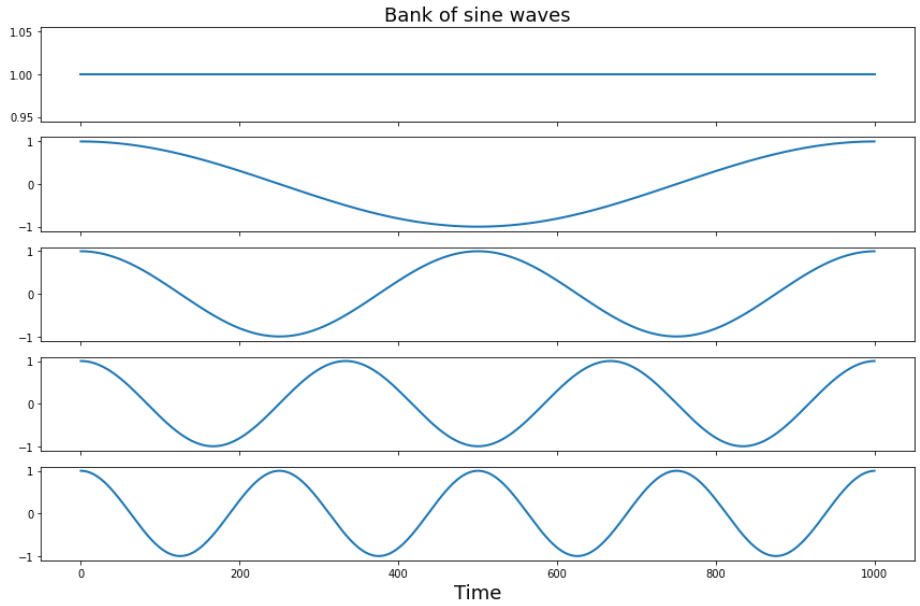


Create a filter bank

Ok, now let's create a bank of n-1 linearly spaced complex sine waves and plot first 5 waves to see their frequencies.

Remember the first basis function is zero frequency component and reflects the mean offset over the entire signal.

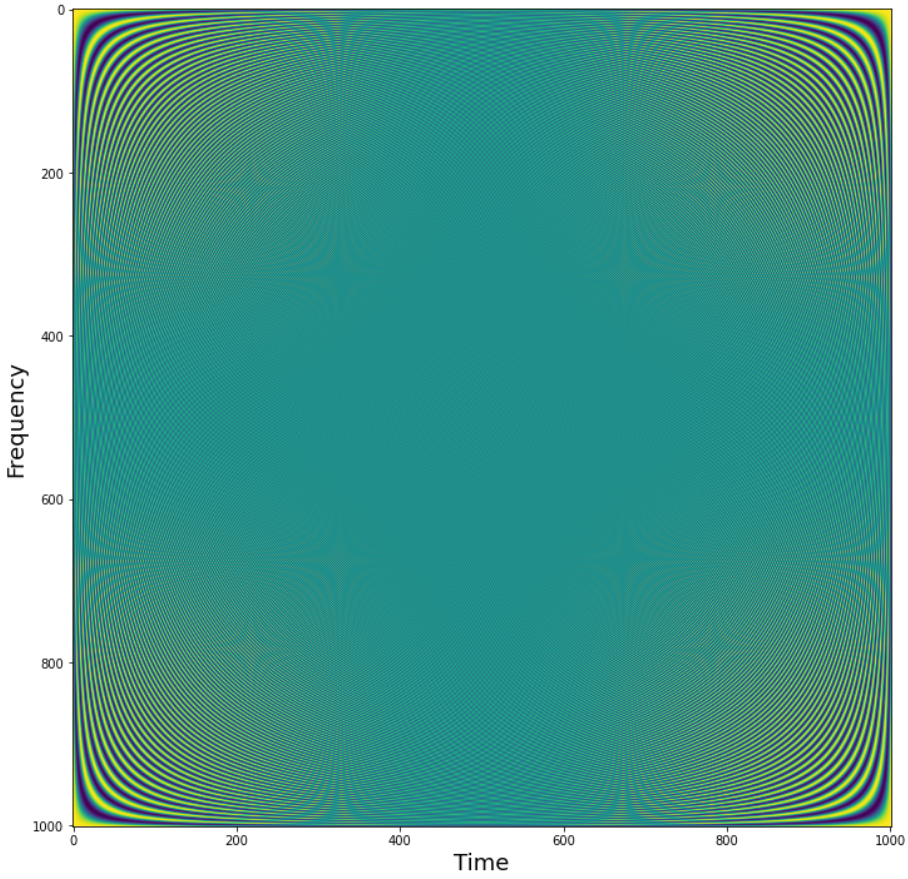
```
import numpy as np
from numpy import exp

time = np.arange(0, len(signal), 1)/len(signal)

sine_waves = []
for i in range(len(signal)):
    sine_waves.append(exp(-1j*2*pi*i*time))
sine_waves = np.array(sine_waves)

f,a = plt.subplots(nrows=5, figsize=(12,8), sharex=True)
for i in range(0,5):
    a[i].plot(sine_waves[i,:], linewidth=2)
a[0].set_title('Bank of sine waves', fontsize=18)
a[i].set_xlabel('Time', fontsize=18)
plt.tight_layout()
```



We can visualize all of the sine waves simultaneously using a heatmap representation. Each row is a different sine wave, and columns reflect time. The intensity of the value is like if the sine wave was coming towards and away rather than up and down. Notice how it looks like that the second half of the sine waves appear to be a mirror image of the first half. This is because the first half contain the *positive* frequencies, while the second half contains the *negative* frequencies. Negative frequencies capture sine waves that travel in reverse order around the complex plane compared to that travel forward. This becomes more relevant with the hilbert transform, but for the purposes of this tutorial we will be ignoring the negative frequencies.

```
plt.figure(figsize = (12, 12))
plt.imshow(np.real(sine_waves))
plt.ylabel('Frequency', fontsize=18)
plt.xlabel('Time', fontsize=18)
```

```
Text(0.5, 0, 'Time')
```



Estimate Fourier Coefficients

Now let's take the dot product of each of the sine wave basis set with our signal to get the fourier coefficients.

We can *scale* the coefficients to be more interpretable by dividing by the number of time points and multiplying by 2. Watch this video if you're interested in a more detailed explanation. Basically, this only needs to be done if you want the amplitude to be in the same units as the original data. In practice, this scaling factor will not change your interpretation of the spectrum.

```
fourier = 2*np.dot(signal, sine_waves)/len(signal)
```

Visualizing Fourier Coefficients

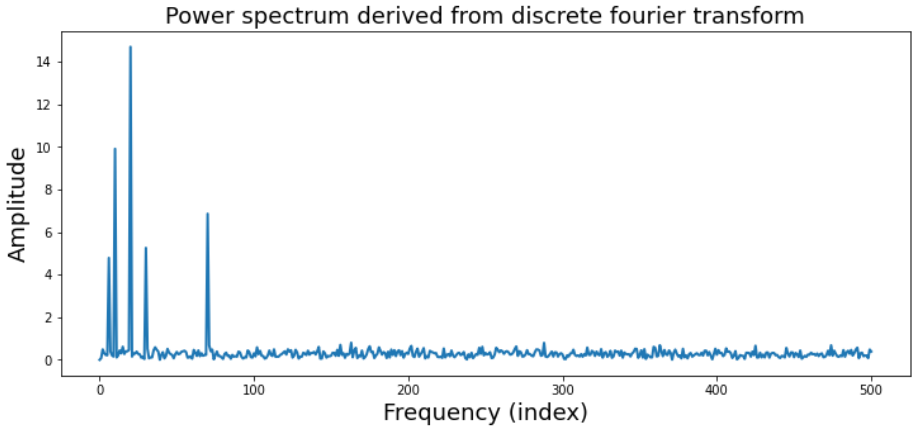
Now that we have computed the fourier transform, we might want to examine the results. The fourier transform provides a 3-D representation of the data including frquency, power, and phase. Typically, the phase information is ignored when plotting the results of a fourier analysis. The traditional way to view the information is plot the data as amplitude on the *y-axis* and frequency on the *x-axis*. We will extract amplitude by taking the absolute value of the fourier coefficients. Remember that we are only focusing on the positive frequencies (the 1st half of the sine wave basis functions).

Here the x axis simply reflects the index of the frequency. The actual frequency is $N/2 + 1$ as we are only able estimate frequencies that are half the sampling frequency, this is called the Nyquist frequency. Also, note that we are only plotting the first half of the frequencies. This is because we are only plotting the *positive* frequencies. We will ignore frequencies above the nyquist frequency (i.e., $\frac{\text{fs}}{2}$), which are called negative frequencies. Watch this video if you'd like more information about why.

Watch this video to hear more about frequencies and zero padding.

```
plt.figure(figsize=(12, 5))
plt.plot(np.abs(fourier[0:int(np.ceil(len(fourier)/2))]), linewidth=2)
plt.xlabel('Frequency (index)', fontsize=18)
plt.ylabel('Amplitude', fontsize=18)
plt.title('Power spectrum derived from discrete fourier transform', fontsize=18)
```

```
Text(0.5, 1.0, 'Power spectrum derived from discrete fourier transform')
```



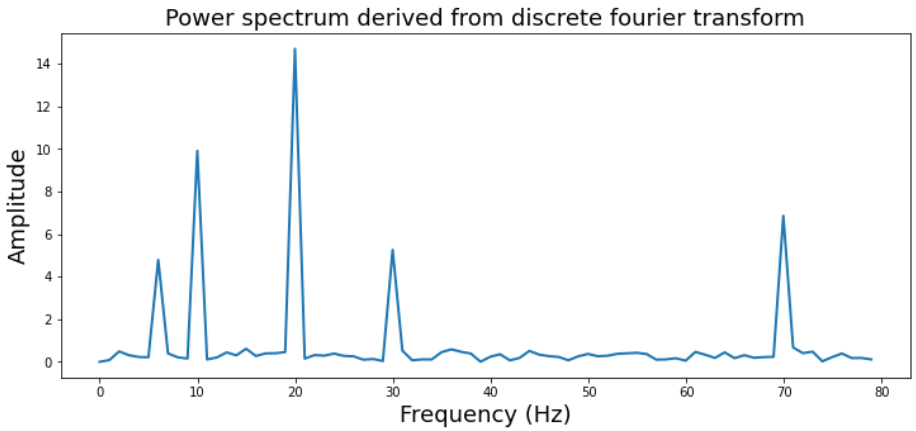
Notice that there are 5 different frequencies that have varying amplitudes. Recall that when we simulated this data we added 5 different sine waves with different frequencies and amplitudes.

`freq = [3, 10, 5 ,15, 35] amplitude = [5, 15, 10, 5, 7]`

Let's zoom in a bit more to see this more clearly and also add the correct frequency labels in Hz. We will use the numpy `fftfreq` function to help convert frequency indices to Hz.

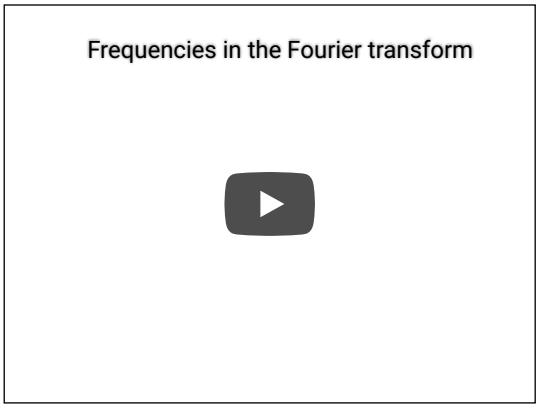
```
from numpy.fft import fftfreq

freq = fftfreq(len(signal),1/sampling_freq)

plt.figure(figsize=(12,5))
plt.plot(freq[:80], np.abs(fourier)[0:80], linewidth=2)
plt.xlabel('Frequency (Hz)', fontsize=18)
plt.ylabel('Amplitude', fontsize=18)
plt.title('Power spectrum derived from discrete fourier transform', fontsize=18)
```

```
Text(0.5, 1.0, 'Power spectrum derived from discrete fourier transform')
```



Ok, now that we've created our own discrete fourier transform, let's learn a few more important details that are important to consider.

```
YouTubeVideo('RHjqvcKVopg')
```



Inverse Fourier Transform

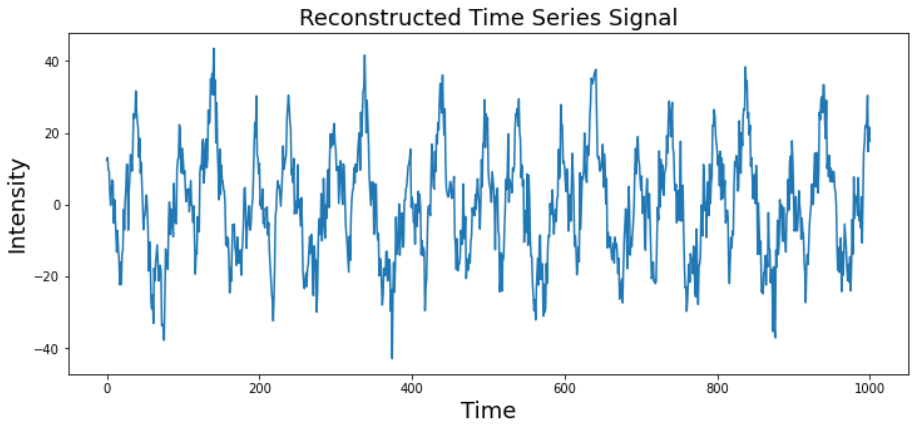
The fourier transform allows you to represent a time series in the frequency domain. This is a lossless operation, meaning that no information in the original signal is lost by the transform. This means that we can reconstruct the original signal by inverting the operation. Thus, we can create a time series with only the frequency domain information using the *inverse fourier transform*. Watch this video if you would like a more in depth explanation.

$$x_k = \sum_{k=0}^{n-1} X_k e^{i2\pi f(k-1)n^{-1}}$$

Notice that we are computing the dot product between the complex sine wave and the fourier coefficients X instead of the time series data x.

```
plt.figure(figsize=(12,5))
plt.plot(np.dot(fourier, sine_waves)/2)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.title('Reconstructed Time Series Signal', fontsize=18)
```

```
Text(0.5, 1.0, 'Reconstructed Time Series Signal')
```

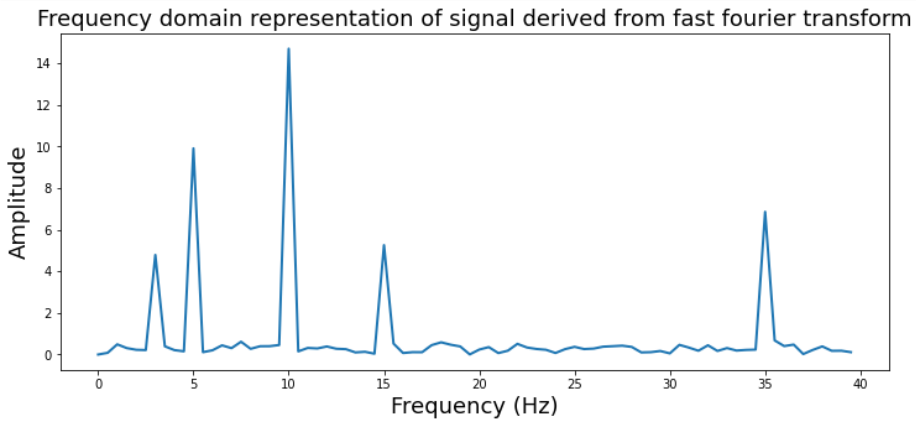


Fast Fourier Transform

The discrete time fourier transform is useful to understand the relationship between the time and frequency domains. However, in practice this method is rarely used as there are more faster and efficient methods to perform this computation. One popular algorithm is called the fast fourier transform (FFT). This function is also in numpy `np.fft.fft`. Don't forget to divide by the number of samples to keep the scaling.

```
from numpy.fft import fft, ifft, fftfreq

fourier_fft = fft(signal)

plt.figure(figsize=(12,5))
plt.plot((np.arange(0,80)/2), 2*np.abs(fourier_fft[0:80])/len(signal), linewidth=2)
plt.ylabel('Amplitude', fontsize=18)
plt.xlabel('Frequency (Hz)', fontsize=18)
plt.title('Frequency domain representation of signal derived from fast fourier
transform', fontsize=18)
```

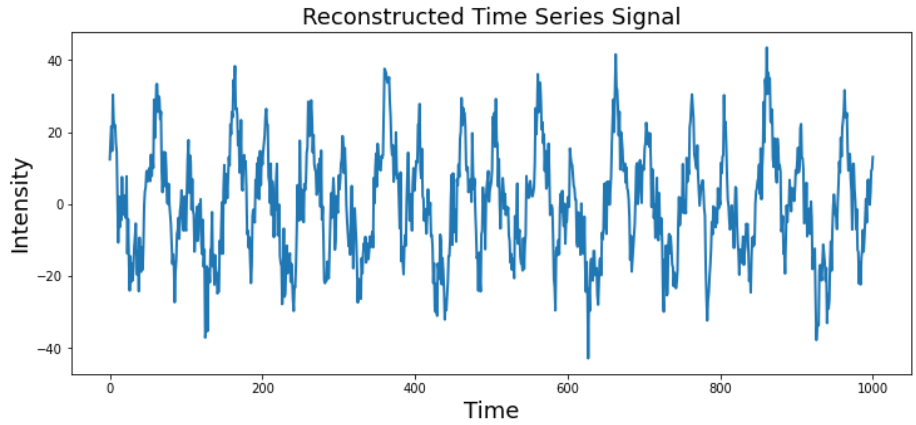
```
Text(0.5, 1.0, 'Frequency domain representation of signal derived from fast fourier
transform')
```



We can also use the `ifft` to perform an inverse fourier transform.

```
plt.figure(figsize=(12, 5))
plt.plot(ifft(fourier_fft), linewidth=2)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.title('Reconstructed Time Series Signal', fontsize=18)
```

```
Text(0.5, 1.0, 'Reconstructed Time Series Signal')
```



Convolution Theorem

Convolution in the time domain is the same multiplication in the frequency domain. This means that time domain convolution computations can be performed much more efficiently in the frequency domain via simple multiplication. (The opposite is also true that multiplication in the time domain is the same as convolution in the frequency domain. Watch this video for an overview of the convolution theorem and convolution in the frequency domain.

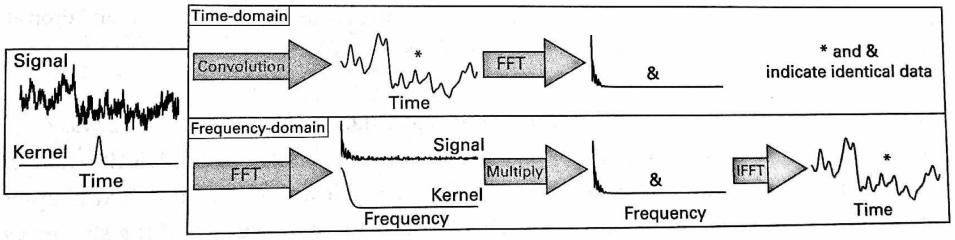


Figure 11.10
Illustration of the convolution theorem and the interchangeability of time-domain convolution and frequency-domain multiplication. The two time series with asterisks are identical, as are the two frequency spectra with ampersands.

Filters

Filters can be classified as finite impulse response (FIR) or infinite impulse response (IIR). These terms describe how a filter responds to a single input impulse. FIR filters have a response that ends at a disrete point in time, while IIR filters have a response that continues indefinitely.

Filters are constructed in the frequency domain and several properties that need to be considers.

- ripple in the pass-band
- attenuation in the stop-band
- steepness of roll-off
- filter order (i.e., length for FIR filters)
- time-domain ringing

In general, there is a frequency by time tradeoff. The sharper something is in frequency, the broader it is in time, and vice versa.

Here we will use IIR butterworth filters as an example.

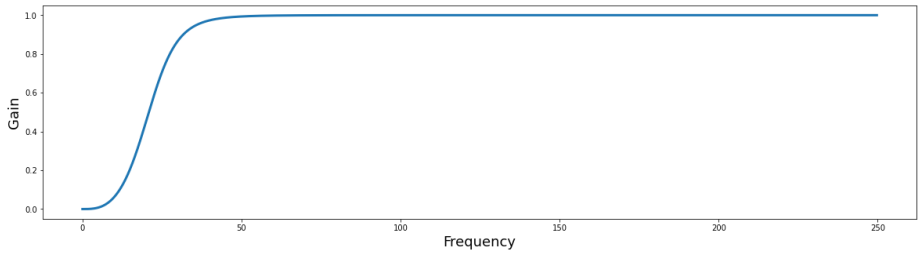
High Pass

High pass filters only allow high frequency signals to remain, effectively *removing* any low frequency information.

Here we will construct a high pass butterworth filter and plot it in frequency space.

Note: this example requires using scipy 1.2.1+.

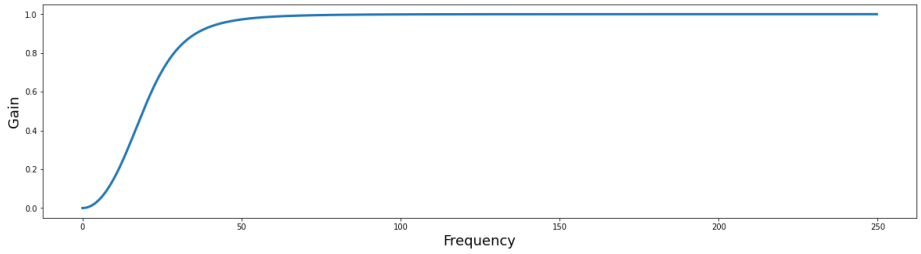
```
from scipy.signal import butter, filtfilt, freqz

filter_order = 3
frequency_cutoff = 25
sampling_frequency = 500

# Create the filter
b, a = butter(filter_order, frequency_cutoff, btype='high', output='ba',
fs=sampling_frequency)

def rad_sample_to_hz(x, fs):
    return (x*fs)/(2*np.pi)

def plot_filter(b, a, fs):
    plt.figure(figsize=(20,5))
    w, h = freqz(b, a, worN=512*2, whole=False)
    plt.plot(rad_sample_to_hz(w, fs), abs(h), linewidth=3)
    plt.ylabel('Gain', fontsize=18)
    plt.xlabel('Frequency', fontsize=18)

plot_filter(b, a, sampling_frequency)
```



Notice how the gain scales from [0,1]? Filters can be multiplied by the FFT of a signal to apply the filter in the frequency domain. When the resulting signal is transformed back in the time domain using the inverse FFT, the new signal will be filtered. This can be much faster than applying filters in the time domain.

The filter_order parameter adjusts the sharpness of the cutoff in the frequency domain. Try playing with different values to see how it changes the filter plot.

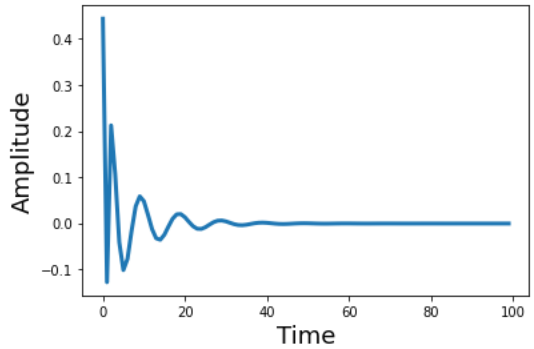
```
filter_order = 2
frequency_cutoff = 25
sampling_frequency = 500


b, a = butter(filter_order, frequency_cutoff, btype='high', output='ba',
fs=sampling_frequency)

plot_filter(b, a, sampling_frequency)
```



What does the filter look like in the temporal domain? Let's take the inverse FFT and plot it to see what it looks like as a kernel in the temporal domain. Notice how changing the filter order adds more ripples in the time domain.

```
from scipy.signal import sosfreqz

filter_order = 8
sos = butter(filter_order, frequency_cutoff, btype='high', output='sos',
fs=sampling_frequency)
w_sos, h_sos = sosfreqz(sos)

plt.plot(ifft(h_sos)[0:100], linewidth=3)
plt.ylabel('Amplitude', fontsize=18)
plt.xlabel('Time', fontsize=18)
```

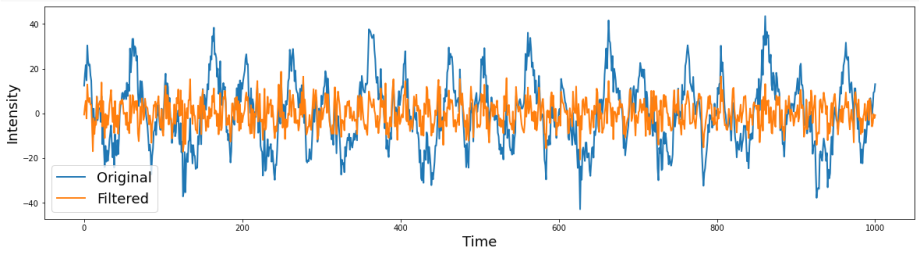
```
Text(0.5, 0, 'Time')
```



Now let's apply the filter to our data. We will be applying the filter to the signal in the time domain using the `filtfilt` function. This is a good default option, even though there are several other functions to apply the filter. `filtfilt` applies the filter forward and then in reverse ensuring that there is zero-phase distortion.

```
filtered = filtfilt(b, a, signal)

plt.figure(figsize=(20,5))
plt.plot(signal, linewidth=2)
plt.plot(filtered, linewidth=2)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.legend(['Original','Filtered'], fontsize=18)
```

```
<matplotlib.legend.Legend at 0x7f95c9a465d0>
```

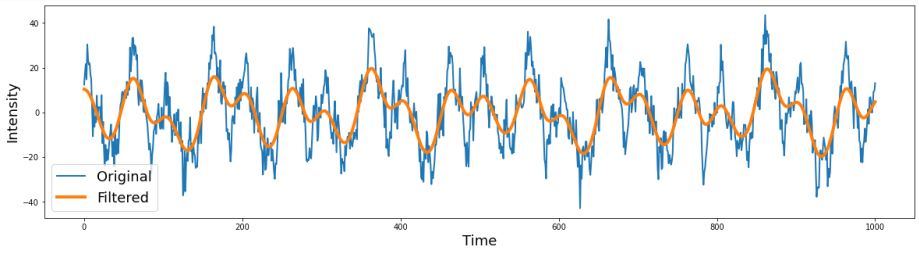


Low Pass

Low pass filters only retain low frequency signals, which *removes* any high frequency information.

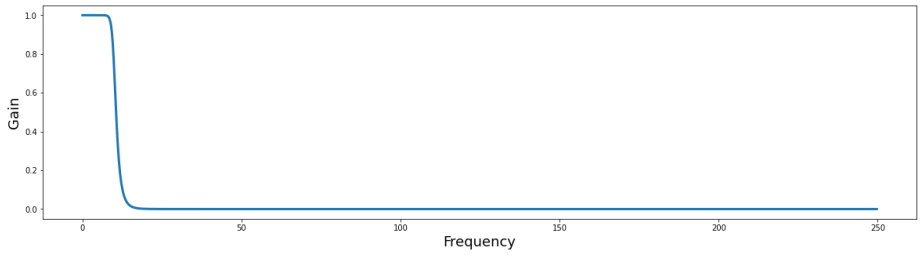
```
from scipy.signal import butter, filtfilt

filter_order = 2
frequency_cutoff = 10
sampling_frequency = 500

# Create the filter
b, a = butter(filter_order, frequency_cutoff, btype='low', output='ba',
fs=sampling_frequency)

# Apply the filter
filtered = filtfilt(b, a, signal)

plt.figure(figsize=(20,5))
plt.plot(signal, linewidth=2)
plt.plot(filtered, linewidth=4)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.legend(['Original','Filtered'], fontsize=18)
```

```
<matplotlib.legend.Legend at 0x7f95b9019c90>
```



What does the filter look like?

```
filter_order = 10
frequency_cutoff = 10
sampling_frequency = 500

# Create the filter
b, a = butter(filter_order, frequency_cutoff, btype='low', output='ba',
fs=sampling_frequency)

plot_filter(b, a, sampling_frequency)
```



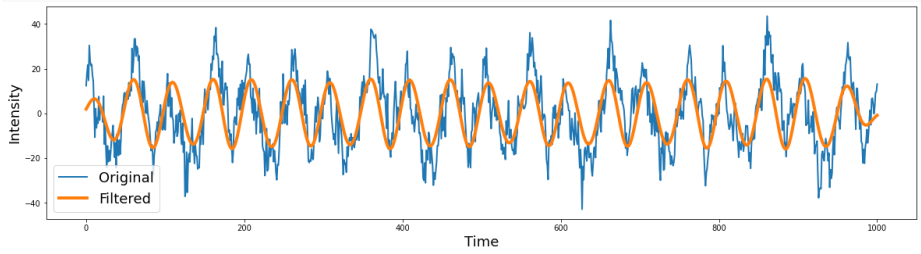
Bandpass

Bandpass filters permit retaining only a specific frequency. Morlet wavelets are an example of a bandpass filter. or example a Morlet wavelet is a gaussian with the peak frequency at the center of a bandpass filter.

Let's try selecting removing specific frequencies

```
filter_order = 2
lowcut = 7
highcut = 13

# Create the filter
b, a = butter(filter_order, [lowcut, highcut], btype='bandpass', output='ba',
fs=sampling_frequency)

# Apply the filter
filtered = filtfilt(b, a, signal)

plt.figure(figsize=(20,5))
plt.plot(signal, linewidth=2)
plt.plot(filtered, linewidth=4)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.legend(['Original','Filtered'], fontsize=18)
```

```
<matplotlib.legend.Legend at 0x7f95d8e26d50>
```

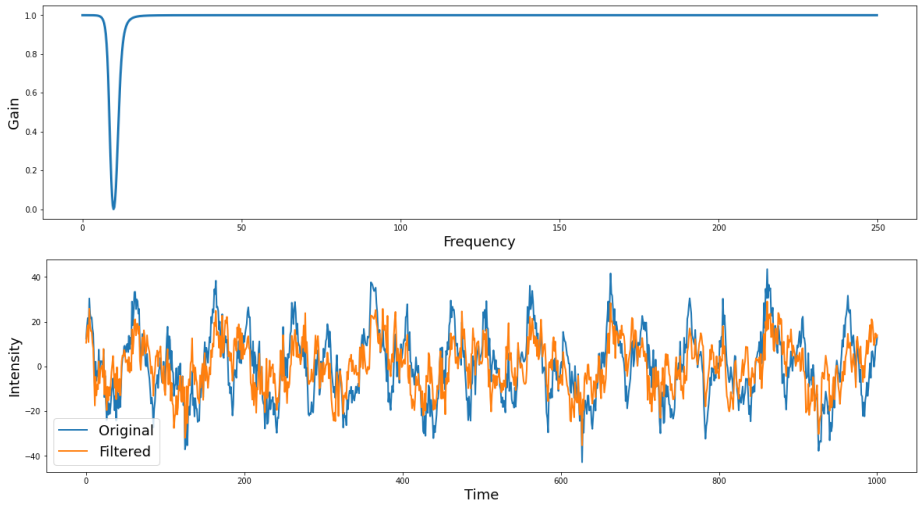


Band-Stop

Bandstop filters remove a specific frequency from the signal

```
filter_order = 2
lowcut = 8
highcut = 12

# Create the filter
b,a = butter(filter_order, [lowcut, highcut], btype='bandstop', output='ba',
fs=sampling_frequency)

# Plot the filter
plot_filter(b, a, sampling_frequency)

# Apply the filter
filtered = filtfilt(b, a, signal)

plt.figure(figsize=(20,5))
plt.plot(signal, linewidth=2)
plt.plot(filtered, linewidth=2)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
plt.legend(['Original','Filtered'], fontsize=18)
```

```
<matplotlib.legend.Legend at 0x7f96088bf390>
```



Exercises

Exercise 1. Create a simulated time series with 7 different frequencies with noise

Exercise 2. Show that you can identify each signal using a FFT

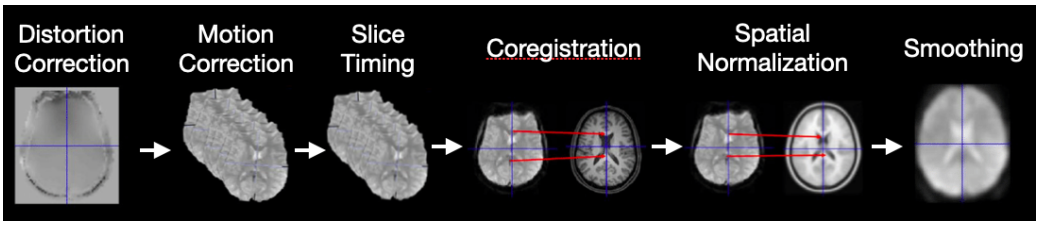
Exercise 3. Remove one frequency with a bandstop filter

Exercise 4. Reconstruct the signal with the frequency removed and compare it to the original

Preprocessing

Written by Luke Chang

Being able to study brain activity associated with cognitive processes in humans is an amazing achievement. However, as we have noted throughout this course, there is an extraordinary amount of noise and a very low levels of signal, which makes it difficult to make inferences about the function of the brain using this BOLD imaging. A critical step before we can perform any analyses is to do our best to remove as much of the noise as possible. The series of steps to remove noise comprise our *neuroimaging data* ***preprocessing*** *pipeline*.



In this lab, we will go over the basics of preprocessing fMRI data using the fmriprep preprocessing pipeline. We will cover:

- Image transformations
- Head motion correction
- Spatial Normalization
- Spatial Smoothing

There are other preprocessing steps that are also common, but not necessarily performed by all labs such as slice timing and distortion correction. We will not be discussing these in depth outside of the videos.

Let's start with watching a short video by Martin Lindquist to get a general overview of the main steps of preprocessing and the basics of how to transform images and register them to other images.

```
from IPython.display import YouTubeVideo

YouTubeVideo('Qc3rRaJWOc4')
```

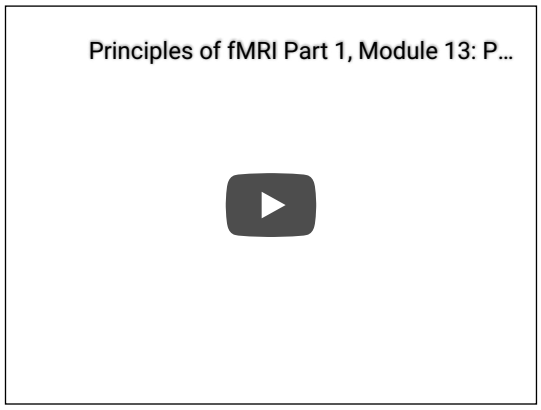


Image Transformations

Ok, now let's dive deeper into how we can transform images into different spaces using linear transformations.

Recall from our introduction to neuroimaging data lab, that neuroimaging data is typically stored in a nifti container, which contains a 3D or 4D matrix of the voxel intensities and also an affine matrix, which provides instructions for how to transform the matrix into another space.

Let's create an interactive plot using ipywidgets so that we can get an intuition for how these affine matrices can be used to transform a 3D image.

We can move the sliders to play with applying rigid body transforms to a 3D cube. A rigid body transformation has 6 parameters: translation in x,y, & z, and rotation around each of these axes. The key thing to remember is that a rigid body transform doesn't allow the image to be fundamentally changed. A full 12 parameter affine transformation adds an additional 3 parameters each for scaling and shearing, which can change the shape of the cube.

Try moving some of the sliders around. Note that the viewer is a little slow. Each time you move a slider it is applying an affine transformation to the matrix and re-plotting.

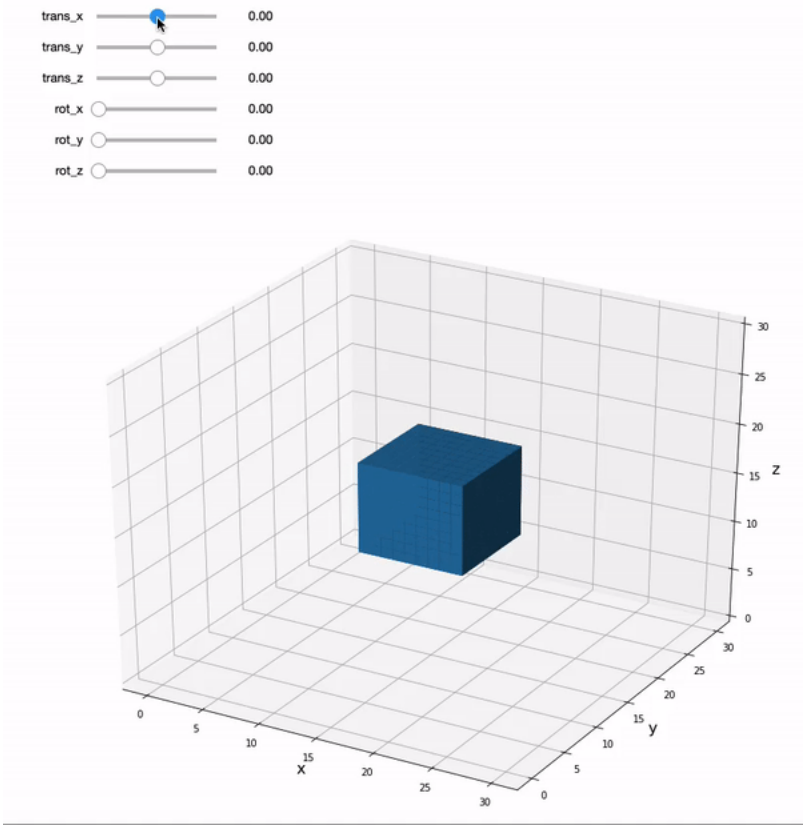
Translation moves the cube in x, y, and z dimensions.

We can also rotate the cube around the x, y, and z axes where the origin is the center point. Continuing to rotate around the point will definitely lead to the cube leaving the current field of view, but it will come back if you keep rotating it.

You'll notice that every time we change the slider and apply a new affine transformation that the cube gets a little distorted with aliasing. Often we need to interpolate the image after applying a transformation to fill in the gaps after applying a transformation. It is important to keep in mind that every time we apply an affine transformation to our images, it is actually not a perfect representation of the original data. Additional steps like reslicing, interpolation, and spatial smoothing can help with this.

```
%matplotlib inline

from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
from nibabel.affines import apply_affine, from_matvec, to_matvec
from scipy.ndimage import affine_transform, map_coordinates
import nibabel as nib
from ipywidgets import interact, FloatSlider

def plot_rigid_body_transformation(trans_x=0, trans_y=0, trans_z=0, rot_x=0, rot_y=0,
rot_z=0):
    '''This plot creates an interactive demo to illustrate the parameters of a rigid
body transformation'''
    fov = 30
    radius = 10
    x, y, z = np.indices((fov, fov, fov))
    cube = ((x > fov//2 - radius//2) & (x < fov//2 + radius//2)) & ((y > fov//2 -
radius//2) & (y < fov//2 + radius//2)) & ((z > fov//2 - radius//2) & (z < fov//2 +
radius//2 ))
    cube = cube.astype(int)

    vec = np.array([trans_x, trans_y, trans_z])

    rot_x = np.radians(rot_x)
    rot_y = np.radians(rot_y)
    rot_z = np.radians(rot_z)
    rot_axis1 = np.array([[1, 0, 0],
                          [0, np.cos(rot_x), -np.sin(rot_x)],
                          [0, np.sin(rot_x), np.cos(rot_x)]])

    rot_axis2 = np.array([[np.cos(rot_y), 0, np.sin(rot_y)],
                          [0, 1, 0],
                          [-np.sin(rot_y), 0, np.cos(rot_y)]])

    rot_axis3 = np.array([[np.cos(rot_z), -np.sin(rot_z), 0],
                          [np.sin(rot_z), np.cos(rot_z), 0],
                          [0, 0, 1]])

    rotation = rot_axis1 @ rot_axis2 @ rot_axis3

    affine = from_matvec(rotation, vec)

    i_coords, j_coords, k_coords = np.meshgrid(range(cube.shape[0]),
range(cube.shape[1]), range(cube.shape[2]), indexing='ij')
    coordinate_grid = np.array([i_coords, j_coords, k_coords])
    coords_last = coordinate_grid.transpose(1, 2, 3, 0)
    transformed = apply_affine(affine, coords_last)
    coords_first = transformed.transpose(3, 0, 1, 2)

    fig = plt.figure(figsize=(15, 12))
    ax = plt.axes(projection='3d')
    ax.voxels(map_coordinates(cube, coords_first))
    ax.set_xlabel('x', fontsize=16)
    ax.set_ylabel('y', fontsize=16)
    ax.set_zlabel('z', fontsize=16)

interact(plot_rigid_body_transformation,
         trans_x=FloatSlider(value=0, min=-10, max=10, step=1),
         trans_y=FloatSlider(value=0, min=-10, max=10, step=1),
         trans_z=FloatSlider(value=0, min=-10, max=10, step=1),
         rot_x=FloatSlider(value=0, min=0, max=360, step=15),
         rot_y=FloatSlider(value=0, min=0, max=360, step=15),
         rot_z=FloatSlider(value=0, min=0, max=360, step=15))
```

```
<function __main__.plot_rigid_body_affine_transformation(trans_x=0, trans_y=0,
trans_z=0, rot_x=0, rot_y=0, rot_z=0)>
```



Ok, so what's going on behind the sliders?

Let's borrow some of the material available in the nibabel documentation to understand how these transformations work.

The affine matrix is a way to transform images between spaces. In general, we have some voxel space coordinate (i, j, k), and we want to figure out how to remap this into a reference space coordinate (x, y, z).

It can be useful to think of this as a coordinate transform function f that accepts a voxel coordinate in the original space as an *input* and returns a coordinate in the *output* reference space:

$$(x, y, z) = f(i, j, k)$$

In theory f could be a complicated non-linear function, but in practice we typically assume that the relationship between (i, j, k) and (x, y, z) is linear (or *affine*), and can be encoded with linear affine transformations comprising translations, rotations, and zooms.

Scaling (zooming) in three dimensions can be represented by a diagonal 3 by 3 matrix. Here's how to zoom the first dimension by p, the second by q and the third by r units:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p & i \\ q & j \\ r & k \end{bmatrix} = \begin{bmatrix} p & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & r \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

A rotation in three dimensions can be represented as a 3 by 3 *rotation matrix* wikipedia rotation matrix. For example, here is a rotation by θ radians around the third array axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

This is a rotation by ϕ radians around the second array axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

A rotation of γ radians around the first array axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

Zoom and rotation matrices can be combined by matrix multiplication.

Here's a scaling of p, q, r units followed by a rotation of θ radians around the third axis followed by a rotation of ϕ radians around the second axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & r \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

This can also be written:

$$M = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & r \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

This might be obvious because the matrix multiplication is the result of applying each transformation in turn on the coordinates output from the previous transformation. Combining the transformations into a single matrix M works because matrix multiplication is associative – $ABCD = (ABC)D$.

A translation in three dimensions can be represented as a length 3 vector to be added to the length 3 coordinate. For example, a translation of a units on the first axis, b on the second and c on the third might be written as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

We can write our function f as a combination of matrix multiplication by some 3 by 3 rotation / zoom matrix M followed by addition of a 3 by 1 translation vector (a, b, c)

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

We could record the parameters necessary for f as the 3 by 3 matrix, M and the 3 by 1 vector (a, b, c).

In fact, the 4 by 4 image *affine array* includes this exact information. If $m_{i,j}$ is the value in row i column j of matrix M, then the image affine matrix A is:

$$A = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & a \\ m_{2,1} & m_{2,2} & m_{2,3} & b \\ m_{3,1} & m_{3,2} & m_{3,3} & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Why the extra row of $[0, 0, 0, 1]$? We need this row because we have rephrased the combination of rotations / zooms and translations as a transformation in *homogenous coordinates* (see wikipedia homogenous coordinates). This is a trick that allows us to put the translation part into the same matrix as the rotations / zooms, so that both translations and rotations / zooms can be applied by matrix multiplication. In order to make this work, we have to add an extra 1 to our input and output coordinate vectors:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & a \\ m_{2,1} & m_{2,2} & m_{2,3} & b \\ m_{3,1} & m_{3,2} & m_{3,3} & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix}$$

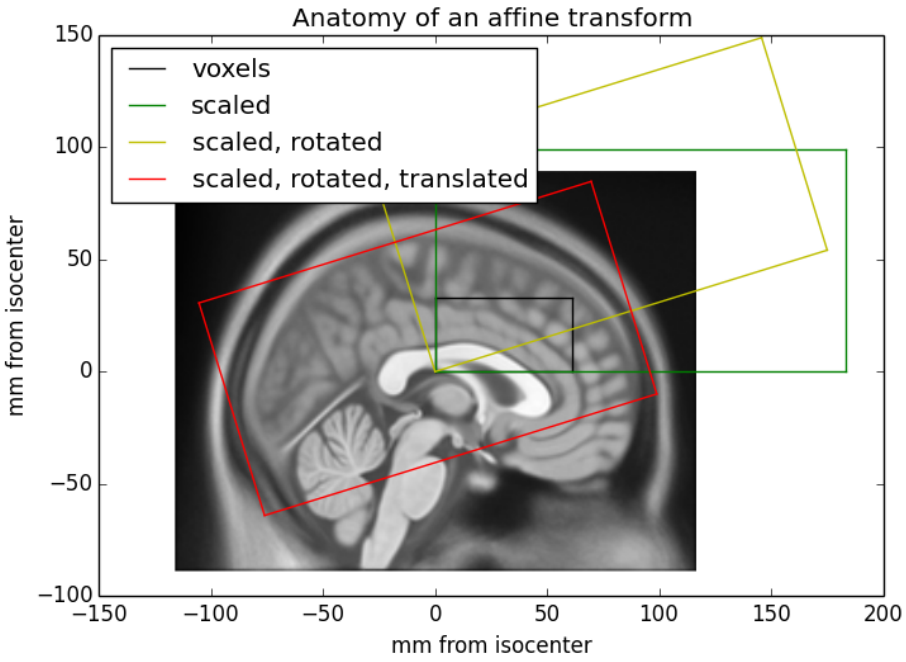
This results in the same transformation as applying M and (a, b, c) separately. One advantage of encoding transformations this way is that we can combine two sets of rotations, zooms, translations by matrix multiplication of the two corresponding affine matrices.

In practice, although it is common to combine 3D transformations using 4 x 4 affine matrices, we usually *apply* the transformations by breaking up the affine matrix into its component M matrix and (a, b, c) vector and doing:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

As long as the last row of the 4 by 4 is $[0, 0, 0, 1]$, applying the transformations in this way is mathematically the same as using the full 4 by 4 form, without the inconvenience of adding the extra 1 to our input and output vectors.

You can think of the image affine as a combination of a series of transformations to go from voxel coordinates to mm coordinates in terms of the magnet isocenter. Here is the EPI affine broken down into a series of transformations, with the results shown on the localizer image:



Applying different affine transformations allows us to rotate, reflect, scale, and shear the image.

Cost Functions

Now that we have learned how affine transformations can be applied to transform images into different spaces, how can we use this to register one brain image to another image?

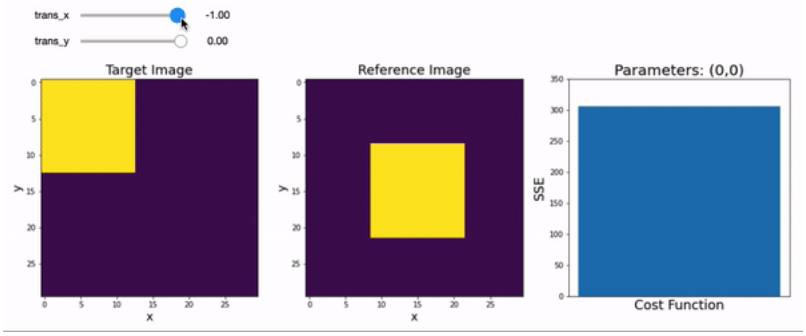
The key is to identify a way to quantify how aligned the two images are to each other. Our visual systems are very good at identifying when two images are aligned, however, we need to create an alignment measure. These measures are often called *cost functions*.

There are many different types of cost functions depending on the types of images that are being aligned. For example, a common cost function is called minimizing the sum of the squared differences and is similar to how regression lines are fit to minimize deviations from the observed data. This measure works best if the images are of the same type and have roughly equivalent signal intensities.

Let's create another interactive plot and find the optimal X & Y translation parameters that minimize the difference between a two-dimensional target image to a reference image.

```
def plot_affine_cost(trans_x=0, trans_y=0):
    '''This function creates an interactive demo to highlight how a cost function works
in image registration.'''
    fov = 30
    radius = 15
    x, y = np.indices((fov, fov))
    square1 = (x < radius-2) & (y < radius-2)
    square2 = ((x > fov//2 - radius//2) & (x < fov//2 + radius//2)) & ((y > fov//2 -
radius//2) & (y < fov//2 + radius//2))
    square1 = square1.astype(float)
    square2 = square2.astype(float)

    vec = np.array([trans_y, trans_x])

    affine = from_matvec(np.eye(2), vec)

    i_coords, j_coords = np.meshgrid(range(square1.shape[0]), range(square1.shape[1]),
indexing='ij')
    coordinate_grid = np.array([i_coords, j_coords])
    coords_last = coordinate_grid.transpose(1, 2, 0)
    transformed = apply_affine(affine, coords_last)
    coords_first = transformed.transpose(2, 0, 1)

    transformed_square = map_coordinates(square1, coords_first)
    f,a = plt.subplots(ncols=3, figsize=(15, 5))
    a[0].imshow(transformed_square)
    a[0].set_xlabel('x', fontsize=16)
    a[0].set_ylabel('y', fontsize=16)
    a[0].set_title('Target Image', fontsize=18)

    a[1].imshow(square2)
    a[1].set_xlabel('x', fontsize=16)
    a[1].set_ylabel('y', fontsize=16)
    a[1].set_title('Reference Image', fontsize=18)

    point_x = deepcopy(trans_x)
    point_y = deepcopy(trans_y)
    sse = np.sum((transformed_square - square2)**2)
    a[2].bar(0, sse)
    a[2].set_ylim([0, 350])
    a[2].set_ylabel('SSE', fontsize=18)
    a[2].set_xlabel('Cost Function', fontsize=18)
    a[2].set_xticks([])
    a[2].set_title(f'Parameters: ({int(trans_x)},{int(trans_y)})', fontsize=20)
    plt.tight_layout()

interact(plot_affine_cost,
         trans_x=FloatSlider(value=0, min=-30, max=0, step=1),
         trans_y=FloatSlider(value=0, min=-30, max=0, step=1))
```

```
<function __main__.plot_affine_cost(trans_x=0, trans_y=0)>
```



You probably had to move the sliders around back and forth until you were able to reduce the sum of squared error to zero. This cost function increases exponentially the further you are away from your target. The process of minimizing (or sometimes maximizing) cost functions to identify the best fitting parameters is called *optimization* and is a concept that is core to fitting models to data across many different disciplines.

Cost Function	Use Case	Example
Sum of Squared Error	Images of same modality and scaling	Two T2* images
Normalized correlation	Images of same modality	two T1 images
Correlation ratio	Any modality	T1 and FLAIR
Mutual information or normalized mutual information	Any modality	T1 and CT
Boundary Based Registration	Images with some contrast across boundaries of interest	EPI and T1

Realignment

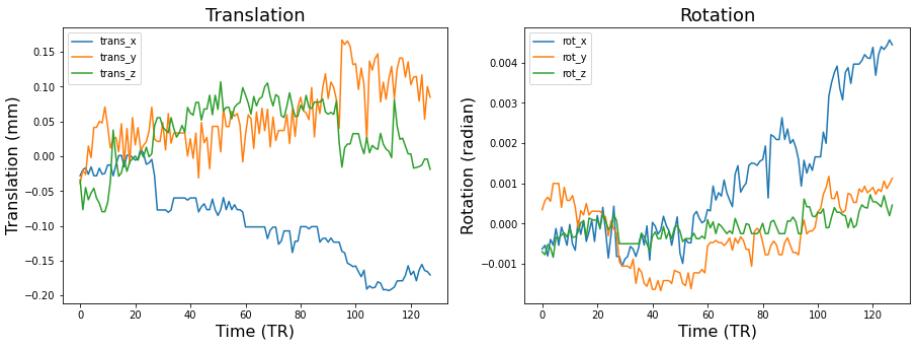
Now let's put everything we learned together to understand how we can correct for head motion in functional images that occurred during a scanning session. It is extremely important to make sure that a specific voxel has the same 3D coordinate across all time points to be able to model neural processes. This of course is made difficult by the fact that participants move during a scanning session and also in between runs.

Realignment is the preprocessing step in which a rigid body transformation is applied to each volume to align them to a common space. One typically needs to choose a reference volume, which might be the first, middle, or last volume, or the mean of all volumes.

Let's look at an example of the translation and rotation parameters after running realignment on our first subject.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from bids import BIDSLayout, BIDSValidator
import os

data_dir = '../data/localizer'
layout = BIDSLayout(data_dir, derivatives=True)

data = pd.read_csv(layout.get(subject='S01', scope='derivatives', extension='.tsv')
[0].path, sep='\t')

f,a = plt.subplots(ncols=2, figsize=(15,5))

data.loc[:,['trans_x','trans_y','trans_z']].plot(ax=a[0])
a[0].set_ylabel('Translation (mm)', fontsize=16)
a[0].set_xlabel('Time (TR)', fontsize=16)
a[0].set_title('Translation', fontsize=18)

data.loc[:,['rot_x','rot_y','rot_z']].plot(ax=a[1])
a[1].set_ylabel('Rotation (radian)', fontsize=16)
a[1].set_xlabel('Time (TR)', fontsize=16)
a[1].set_title('Rotation', fontsize=18)
```

```
Text(0.5, 1.0, 'Rotation')
```



Don't forget that even though we can approximately put each volume into a similar position with realignment that head motion always distorts the magnetic field and can lead to nonlinear changes in signal intensity that will not be addressed by this procedure. In the resting-state literature, where many analyses are based on functional

connectivity, head motion can lead to spurious correlations. Some researchers choose to exclude any subject that moved more than certain amount. Other's choose to remove the impact of these time points in their data through removing the volumes via *scrubbing* or modeling out the volume with a dummy code in the first level general linear models.

Spatial Normalization

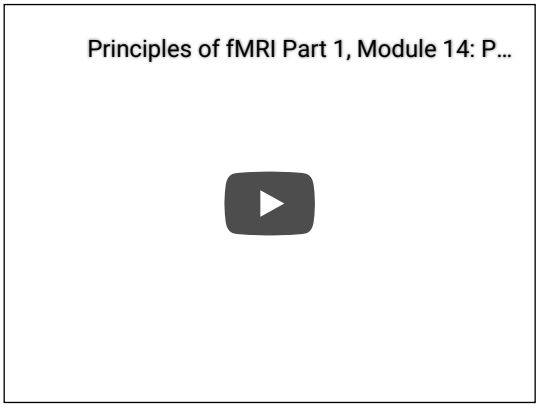
There are several other preprocessing steps that involve image registration. The main one is called *spatial normalization*, in which each subject's brain data is warped into a common stereotactic space. Talaraich is an older space, that has been subsumed by various standards developed by the Montreal Neurological Institute.

There are a variety of algorithms to warp subject data into stereotactic space. Linear 12 parameter affine transformation have been increasingly been replaced by more complicated nonlinear normalizations that have hundreds to thousands of parameters.

One nonlinear algorithm that has performed very well across comparison studies is *diffeomorphic registration*, which can also be inverted so that subject space can be transformed into stereotactic space and back to subject space. This is the core of the ANTs algorithm that is implemented in fmriprep. See this overview for more details.

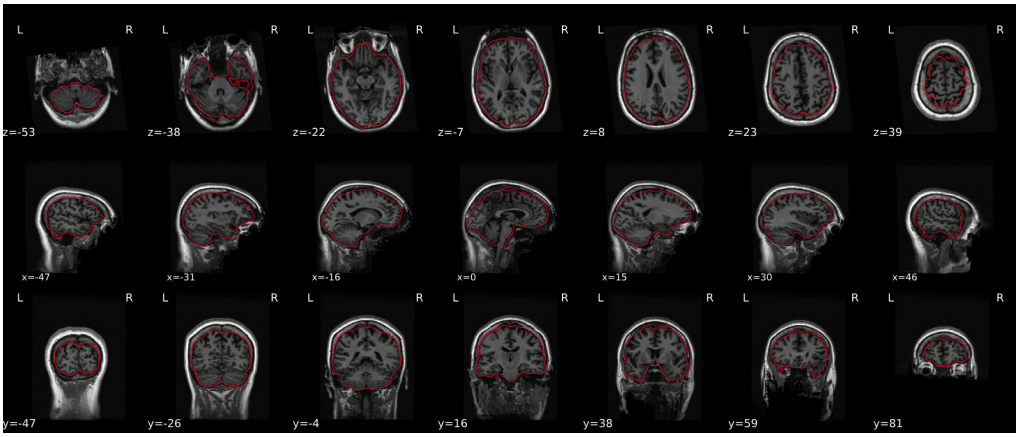
Let's watch another short video by Martin Lindquist and Tor Wager to learn more about the core preprocessing steps.

```
YouTubeVideo('qamRGWSC-6g')
```



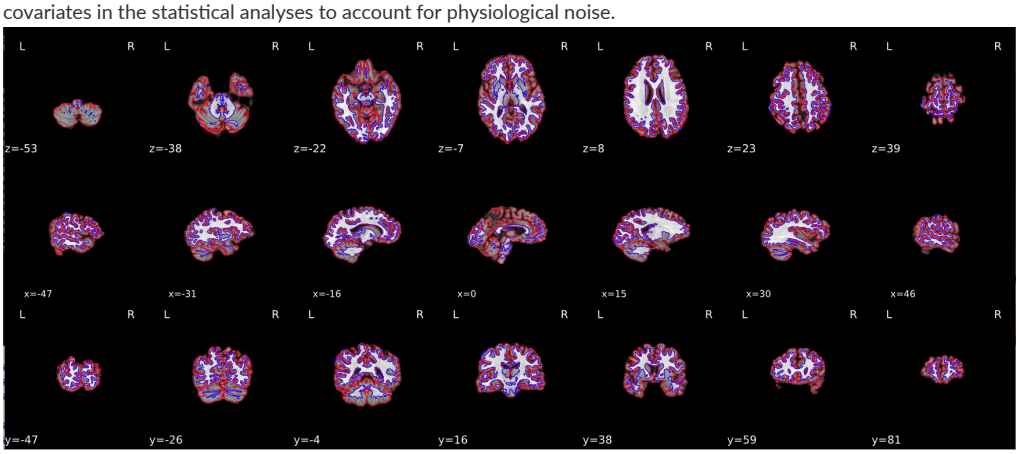
There are many different steps involved in the spatial normalization process and these details vary widely across various imaging software packages. We will briefly discuss some of the steps involved in the anatomical preprocessing pipeline implemented by fMRIprep and will be showing example figures from the output generated by the pipeline.

First, brains are extracted from the skull and surrounding dura mater. You can check and see how well the algorithm performed by examining the red outline.



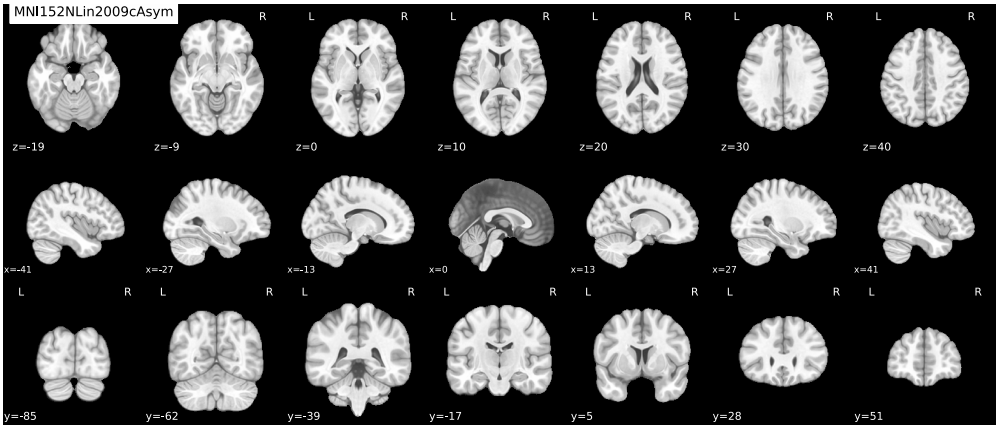
Next, the anatomical images are segmented into different tissue types, these tissue maps are used for various types of analyses, including providing a grey matter mask to reduce the computational time in estimating statistics. In addition, they provide masks to aid in extracting average activity in CSF, or white matter, which might be used as

covariates in the statistical analyses to account for physiological noise.



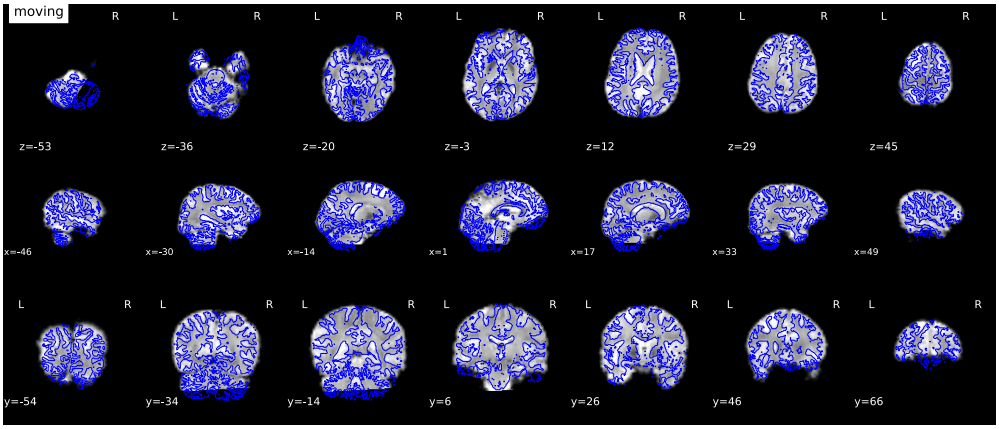
Spatial normalization of the anatomical T1w reference

fmriprep uses the ANTs to perform nonlinear spatial normaliziation. It is easy to check to see how well the algorithm performed by viewing the results of aligning the T1w reference to the stereotactic reference space. Hover on the panels with the mouse pointer to transition between both spaces. We are using the MNI152NLin2009cAsym template.



Alignment of functional and anatomical MRI data

Next, we can evaluate the quality of alignment of the functional data to the anatomical T1 image. FSL `flirt` was used to generate transformations from EPI-space to T1w-space - The white matter mask calculated with FSL `fast` (brain tissue segmentation) was used for BBR. Note that Nearest Neighbor interpolation is used in the reportlets in order to highlight potential spin-history and other artifacts, whereas final images are resampled using Lanczos interpolation. Notice these images are much blurrier and show some distortion compared to the T1s.



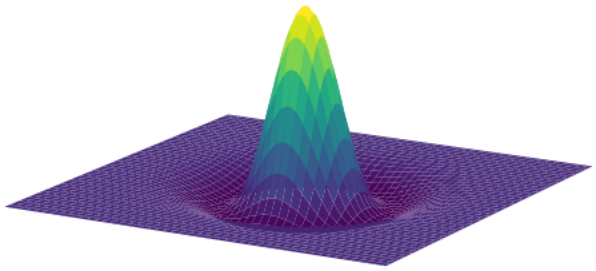
Spatial Smoothing

The last step we will cover in the preprocessing pipeline is *spatial smoothing*. This step involves applying a filter to the image, which removes high frequency spatial information. This step is identical to convolving a kernel to a 1-D signal that we covered in the Signal Processing Basics lab, but the kernel here is a 3-D Gaussian kernel. The amount of smoothing is determined by specifying the width of the distribution (i.e., the standard deviation) using the Full Width at Half Maximum (FWHM) parameter.

Why we would want to decrease our image resolution with spatial smoothing after we tried very hard to increase our resolution at the data acquisition stage? This is because this step may help increase the signal to noise ratio by reducing the impact of partial volume effects, residual anatomical differences following normalization, and other aliasing from applying spatial transformation.

Here is what a 3D gaussian kernel looks like.

```
def plot_gaussian(sigma=2, kind='surface', cmap='viridis', linewidth=1, **kwargs):
    '''Generates a 3D matplotlib plot of a Gaussian distribution'''
    mean=0
    domain=10
    x = np.arange(-domain + mean, domain + mean, sigma/10)
    y = np.arange(-domain + mean, domain + mean, sigma/10)
    x, y = np.meshgrid(x, x)
    r = (x ** 2 + y ** 2) / (2 * sigma ** 2)
    z = 1 / (np.pi * sigma ** 4) * (1 - r) * np.exp(-r)

    fig = plt.figure(figsize=(12, 6))

    ax = plt.axes(projection='3d')
    if kind=='wire':
        ax.plot_wireframe(x, y, z, cmap=cmap, linewidth=linewidth, **kwargs)
    elif kind=='surface':
        ax.plot_surface(x, y, z, cmap=cmap, linewidth=linewidth, **kwargs)
    else:
        NotImplemented

    ax.set_xlabel('x', fontsize=16)
    ax.set_ylabel('y', fontsize=16)
    ax.set_zlabel('z', fontsize=16)
    plt.axis('off')

plot_gaussian(kind='surface', linewidth=1)
```



fmriprep

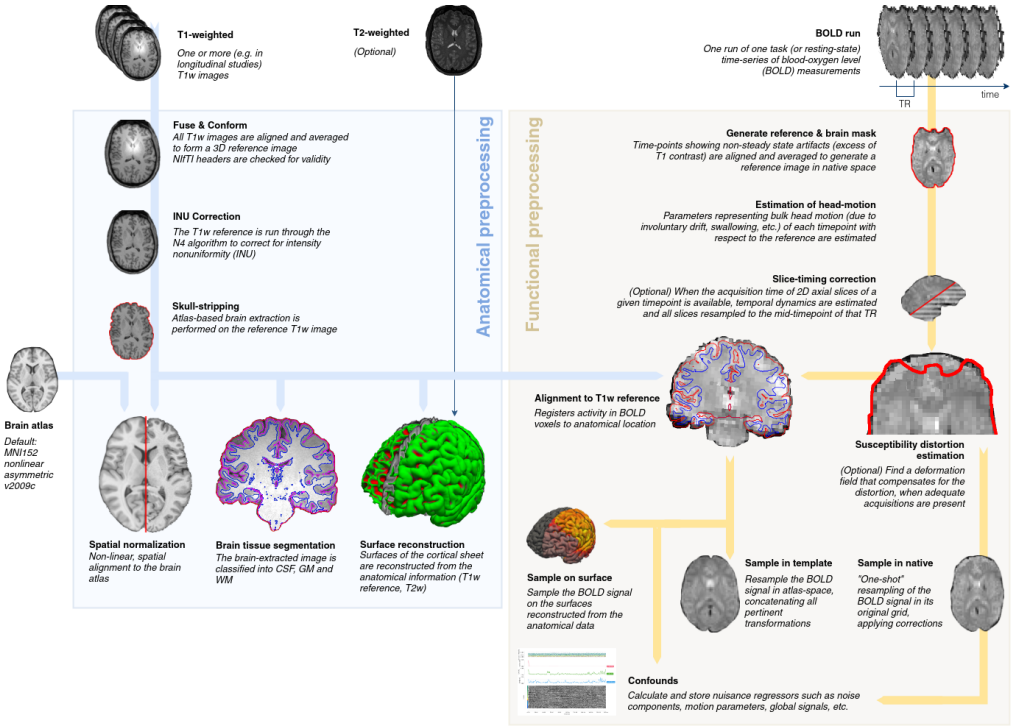
Throughout this lab and course, you have frequently heard about fmriprep, which is a functional magnetic resonance imaging (fMRI) data preprocessing pipeline that was developed by a team at the Center for Reproducible Research led by Russ Poldrack and Chris Gorgolewski. Fmriprep was designed to provide an easily accessible, state-of-the-art interface that is robust to variations in scan acquisition protocols, requires minimal user input, and provides easily interpretable and comprehensive error and output reporting. Fmriprep performs basic processing steps (coregistration, normalization, unwarping, noise component extraction, segmentation, skullstripping etc.) providing outputs that are ready for data analysis.

fmriprep was built on top of nipype, which is a tool to build preprocessing pipelines in python using graphs. This provides a completely flexible way to create custom pipelines using any type of software while also facilitating easy parallelization of steps across the pipeline on high performance computing platforms. Nipype is completely flexible, but has a fairly steep learning curve and is best for researchers who have strong opinions about how they want to

preprocess their data, or are working with nonstandard data that might require adjusting the preprocessing steps or parameters. In practice, most researchers typically use similar preprocessing steps and do not need to tweak the pipelines very often. In addition, many researchers do not fully understand how each preprocessing step will impact their results and would prefer if somebody else picked suitable defaults based on current best practices in the literature. The fmriprep pipeline uses a combination of tools from well-known software packages, including FSL_, ANTs_, FreeSurfer_ and AFNI_. This pipeline was designed to provide the best software implementation for each state of preprocessing, and is quickly being updated as methods evolve and bugs are discovered by a growing user base.

This tool allows you to easily do the following:

- Take fMRI data from raw to fully preprocessed form.
- Implement tools from different software packages.
- Achieve optimal data processing quality by using the best tools available.
- Generate preprocessing quality reports, with which the user can easily identify outliers.
- Receive verbose output concerning the stage of preprocessing for each subject, including meaningful errors.
- Automate and parallelize processing steps, which provides a significant speed-up from typical linear, manual processing.
- More information and documentation can be found at https://fmriprep.readthedocs.io/



Running fmriprep

Running fmriprep is a (mostly) trivial process of running a single line in the command line specifying a few choices and locations for the output data. One of the annoying things about older neuroimaging software that was developed by academics is that the packages were developed using many different development environments and on different operating systems (e.g., unix, windows, mac). It can be a nightmare getting some of these packages to install on more modern computing systems. As fmriprep uses many different packages, they have made it much easier to circumvent the time-consuming process of installing many different packages by releasing a docker container that contains everything you need to run the pipeline.

Unfortunately, our AWS cloud instances running our jupyter server are not equipped with enough computational resources to run fmriprep at this time. However, if you're interested in running this on your local computer, here is the code you could use to run it in a jupyter notebook, or even better in the command line on a high performance computing environment.

```
import os
base_dir = '/Users/lukechang/Dropbox/Dartbrains/Data'
data_path = os.path.join(base_dir, 'localizer')
output_path = os.path.join(base_dir, 'preproc')
work_path = os.path.join(base_dir, 'work')

sub = 'S01'
subs = [f'S{x:0>2d}' for x in range(10)]
for sub in subs:
    !fmriprep-docker {data_path} {output_path} participant --participant_label sub-{sub} --
write-graph --fs-no-reconall --notrack --fs-license-file
~/Dropbox/Dartbrains/License/license.txt --work-dir {work_path}
```

Quick primer on High Performance Computing

We could run fmriprep on our computer, but this could take a long time if we have a lot of participants. Because we have a limited amount of computational resources on our laptops (e.g., cpus, and memory), we would have to run each participant sequentially. For example, if we had 50 participants, it would take 50 times longer to run all participants than a single one.

Imagine if you had 50 computers and ran each participant separate at the same time in parallel across all of the computers. This would allow us to run 50 participants in the same amount of time as a single participant. This is the basic idea behind high performance computing, which contains a cluster of many computers that have been installed in racks. Below is a picture of what Dartmouth's Discovery cluster looks like:

A cluster is simply a collection of nodes. A node can be thought of as an individual computer. Each node contains processors, which encompass multiple cores. Discovery contains 3000+ cores, which is certainly a lot more than your laptop!

In order to submit a job, you can create a Portable Batch System (PBS) script that sets up the parameters (e.g., how much time you want your script to run, specifying directory to run, etc) and submits your job to a queue.

NOTE: For this class, we will only be using the jupyterhub server, but if you end up working in a lab in the future, you will need to request access to the *discovery* system using this link.

fmriprep output

You can see a summary of the operations fmriprep performed by examining the .html files in the `derivatives/fmriprep` folder within the `localizer` data directory.

We will load the first subject's output file. Spend some time looking at the outputs and feel free to examine other subjects as well. Currently, the first 10 subjects should be available on the jupyterhub.

```
from IPython.display import HTML

HTML('sub-S01.html')
```

../data/localizer/derivatives/fmriprep/sub-01.html

Limitations of fmriprep

In general, we recommend using this pipeline if you want a sensible default. Considerable thought has gone into selecting reasonable default parameters and selecting preprocessing steps based on best practices in the field (as determined by the developers). This is not necessarily the case for any of the default settings in any of the more conventional software packages (e.g., spm, fsl, afni, etc).

However, there is an important tradeoff in using this tool. On the one hand, it's nice in that it is incredibly straightforward to use (one line of code!), has excellent documentation, and is actively being developed to fix bugs and improve the overall functionality. There is also a growing user base to ask questions. Neurostars is an excellent forum to post questions and learn from others. On the other hand, fmriprep, is unfortunately in its current state not easily customizable. If you disagree with the developers about the order or specific preprocessing steps, it is very difficult to modify. Future versions will hopefully be more modular and easier to make custom pipelines. If you need this type of customizability we strongly recommend using nipype over fmriprep.

In practice, it's alway a little bit finicky to get everything set up on a particular system. Sometimes you might run into issues with a specific missing file like the freesurfer license even if you're not using it. You might also run into issues with the format of the data that might have some conflicts with the bids-validator. In our experience, there is always some frustrations getting this to work, but it's very nice once it's done.

Exercises

Exercise 1. Inspect HTML output of other participants.

For this exercise, you will need to navigate to the derivatives folder containing the fmriprep preprocessed data `../data/data/localizer/derivatives/fmriprep` and inspect the html output of other subjects (ie., not 'S01'). Did the preprocessing steps works? are there any issues with the data that we should be concerned about?

Introduction to the General Linear Model

Written by Luke Chang

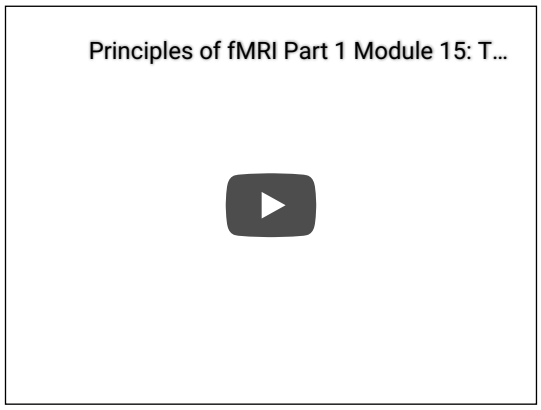
This tutorial provides an introduction for how the general linear model (GLM) can be used to make inferences about brain responses in a single subject. We will explore the statistics in the context of a simple hypothetical experiment using simulated data.

In this lab we will cover:

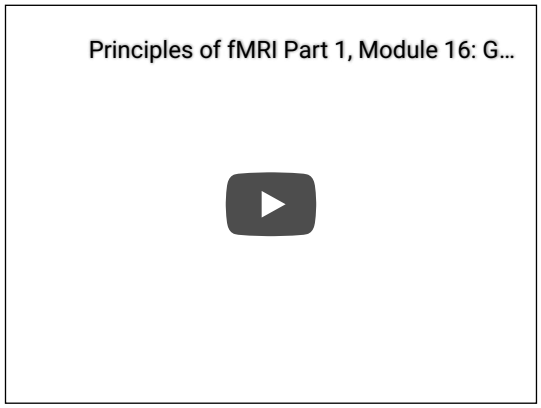
- How to use a GLM to test psychological hypotheses.
- Simulating brain data
- Estimating GLM using ordinary least squares
- Calculating Standard Errors
- Contrast Basics

Let's start by watching two short videos introducing the general linear model by Tor Wager and how this can be applied to fMRI.

```
from IPython.display import YouTubeVideo

YouTubeVideo('GDkLQuV4he4')
```




```
YouTubeVideo('0yLKMb9FNhg')
```



Are you ready for this?

This lab assumes that you have some basic background knowledge in statistics from an introductory course. If you are already feeling overwhelmed from Tor Wager's videos and think you might need to slow down and refresh some basic concepts and lingo, I highly encourage you to watch Jeannette Mumford's crash course in statistics. These are certainly not required, but she is a wonderful teacher and watching her videos will provide an additional explanation of the core concepts needed to understand the GLM. You could watch these in one sitting, or go back and forth with working through the notebooks. There is so much to know in statistics and people can often feel lost because the concepts are certainly not intuitive. For example, even though advanced statistics have been an important part of my own work, I still find it helpful to periodically revisit core concepts. In general, I find that learning neuroimaging is an iterative process. In the beginning, it is important to get a broad understanding of the key steps and how neuroimaging can be used to make inferences, but as you progress in your training you will have plenty of opportunities to zoom into specific steps to learn more about particular details and nuances that you may not have fully appreciated the first time around.

- Basic statistics terminology This video gently introduces some of the key concepts that provide the foundation for statistics.
- Simple Linear Regression This video explains how a regression works using a single variable.
- Matrix Algebra Basics This video provides the background linear algebra needed for understanding the GLM.
- Multiple Linear Regression This video explains how multiple regression works using linear algebra.

- Hypothesis Testing This video covers the basics of hypothesis testing.
- Contrasts in Linear Models This video provides an overview of how to test hypotheses using contrasts in the context of the GLM.
- Intepreting Regression Parameters This video covers how to interpret the results from a regression analysis.
- Mean Centering Regressors This video covers a more subtle detail of why you might consider mean centering your continuour regression variables.

Ok, let's get started. First, we will need to import all of the modules used in this tutorial.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.stats import regress
from nltools.external import glover_hrf
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-
packages/sklearn/utils/deprecation.py:144: FutureWarning: The
sklearn.linear_model.base module is  deprecated in version 0.22 and will be removed
in version 0.24. The corresponding classes / functions should instead be imported
from sklearn.linear_model. Anything that cannot be imported from sklearn.linear_model
is now part of the private API.
  warnings.warn(message, FutureWarning)
```

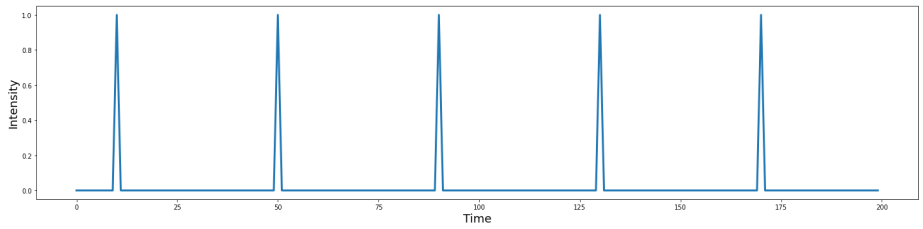
Simulate a voxel time course

To generate an intuition for how we use the GLM to make inferences in fMRI data analysis, we will simulate a time series for a single voxel. A simulation means that we will be generating synthetic data that will resemble real data. However, because we know the ground truth of the signal, we can evaluate how well we can recover the true signal using a general linear model. Throughout this course, we frequently rely on simulations to gain an intuition for how a particular preprocessing step or statistic works. This is important because it reinforces the assumptions behind the operation (which are rarely met in real data), and also provides a method to learn how to answer your own questions by generating your own simulations.

Imagine that we are interested in identifying which region of the brain is involved in processing faces. To explore this question, we could show participants a bunch of different types of faces. Each presentation of a face will be a *trial*. Let's simulate what a design might look like with 5 face trials.

First, we will need to specify the number of volumes in the time series. Then we need to specify the timepoint, in which a face is presented.

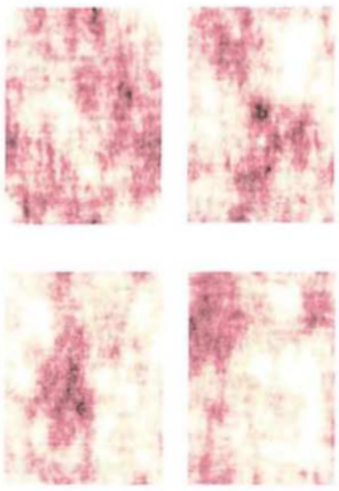
```
n_tr = 200
n_trial = 5
face = np.zeros(n_tr)
face[np.arange(10, n_tr, int(n_tr/n_trial))] = 1

def plot_timeseries(data, labels=None, linewidth=3):
    '''Plot a timeseries

    Args:
        data: (np.ndarray) signal varying over time, where each column is a different
signal.
        labels: (list) labels which need to correspond to the number of columns.
        linewidth: (int) thickness of line
    '''
    plt.figure(figsize=(20,5))
    plt.plot(data, linewidth=linewidth)
    plt.ylabel('Intensity', fontsize=18)
    plt.xlabel('Time', fontsize=18)
    plt.tight_layout()
    if labels is not None:
        if len(labels) != data.shape[1]:
            raise ValueError('Need to have the same number of labels as columns in
data.')
        plt.legend(labels, fontsize=18)

plot_timeseries(face)
```



We now have 5 events where a face is shown for 2 seconds (i.e., one TR). If we scanned someone with this design, we might expect to see any region involved in processing faces increase in activation around the time of the face presentation. How would we know which of these regions, if any, *selectively* process faces? Many of the regions we would observe are likely involved in processing *any* visual stimulus, and not specifically faces.

To rule out this potential confound, we would need at least one other condition that would serve as a visual control. Something that might have similar properties to a face, but isn't a face.

One possibility is to create a visual stimulus that has all of the same visual properties in terms of luminance and color, but no longer resembles a face. Here is an example of the same faces that have been Fourier transformed, phase-scrambled, and inverse Fourier transformed. These pictures have essentially identical low level visual properties, but are clearly not faces.



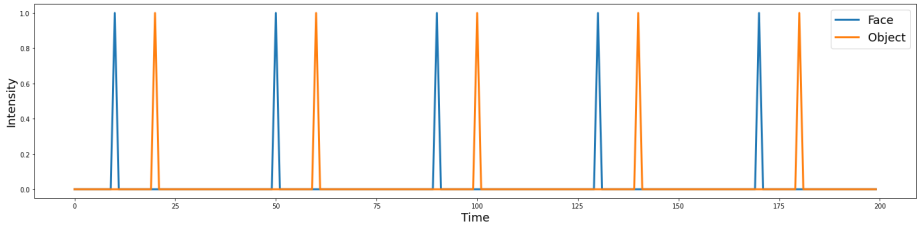
However, one might argue that faces are a type of object, and regions that are involved in higher visual processing such as object recognition might not be selective to processing faces. To rule out this possibility, we would need to add an additional visual control such as objects.

Both of these conditions could serve as a different type of visual control. To keep things simple, let's start with pictures of objects as it controls for low level visual features, but also more complex object processing.

To demonstrate that a region is processing faces and not simply lower level visual properties or objects more generally, we can search for regions that are selectively more activated in response to viewing faces relative to objects. This is called a *contrast* and is the basic principle of the subtraction method for controlling for potential experimental confounds. Because BOLD fMRI is a relative and not absolute measure of brain activity, the subtraction method is a key aspect of experimental design.

Figures are from Huettel, Song, & McCarthy (2008)

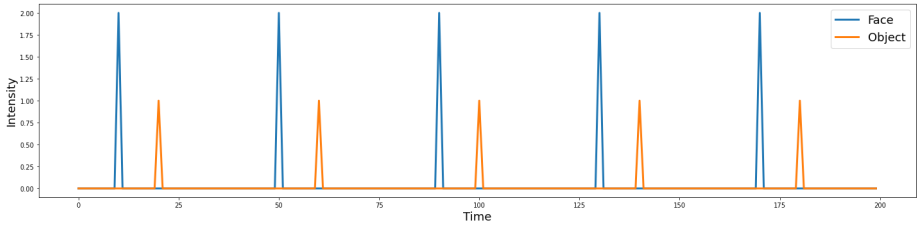
```
n_tr = 200
n_trial = 5
face = np.zeros(n_tr)
face[np.arange(10, n_tr, int(n_tr/n_trial))] = 1
obj = np.zeros(n_tr)
obj[np.arange(20, n_tr, int(n_tr/n_trial))] = 1
voxel = np.vstack([face,obj]).T

plot_timeseries(voxel, labels=['Face', 'Object'])
```



Let's imagine that in a voxel processing face specific information we might expect to see a larger activation in response to faces. Maybe two times bigger?

In our simulation, these two values are parameters we are specifying to generate the data. Specifically they refer to the amplitude of the response to Faces and Houses within a particular region of the brain.

```
n_tr = 200
n_trial = 5
face_intensity = 2
object_intensity = 1

face = np.zeros(n_tr)
face[np.arange(10, n_tr, int(n_tr/n_trial))] = face_intensity
obj = np.zeros(n_tr)
obj[np.arange(20, n_tr, int(n_tr/n_trial))] = object_intensity
voxel = np.vstack([face,obj]).T

plot_timeseries(voxel, labels=['Face', 'Object'])
```

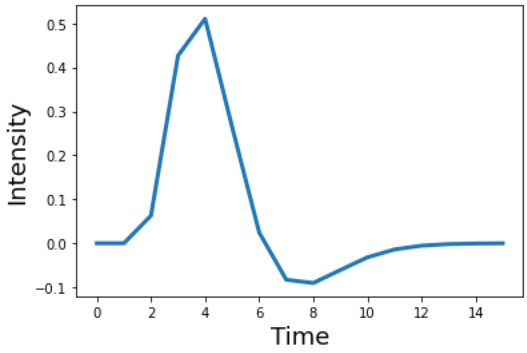


Ok, now we have two conditions that are alternating over time.

We know that the brain has a delayed hemodynamic response to events that has a particular shape, so we will need to convolve these events with an appropriate HRF function. Here, we will use the double-gamma HRF function.

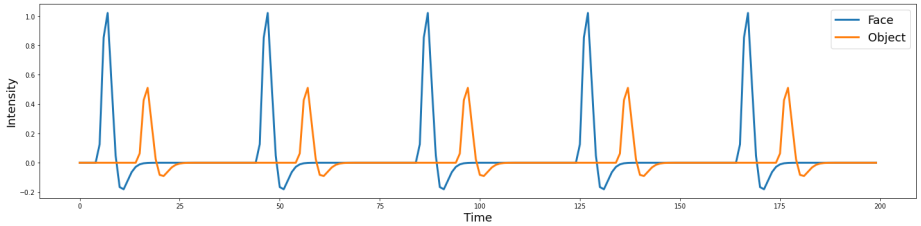
```
tr = 2
hrf = glover_hrf(tr, oversampling=1)
plt.plot(hrf, linewidth=3)
plt.ylabel('Intensity', fontsize=18)
plt.xlabel('Time', fontsize=18)
```

```
Text(0.5, 0, 'Time')
```



We will use `np.convolve` from numpy to perform the convolution. The length of the convolved data will be the length of the time series plus the length of the kernel minus 1. To make sure everything is the same length, we will chop off the extra time off the convolved time series using `mode='same'`.

```
face_conv = np.convolve(face, hrf, mode='same')
obj_conv = np.convolve(obj, hrf, mode='same')
voxel_conv = np.vstack([face_conv, obj_conv]).T

plot_timeseries(voxel_conv, labels=['Face', 'Object'])
```



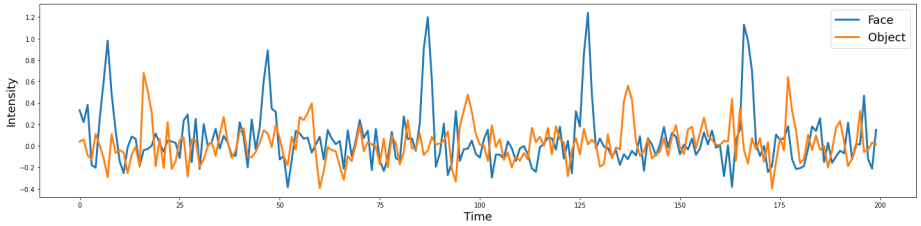
While this might reflect the expected HRF response to a single event, real data is much noiser. It is easy to add different types of noise. For example, there might be a low frequency drift, autocorrelation, or possibly some aliased physiological artifacts.

For now, let's start with something simple, like independent white noise drawn from a random Gaussian distribution

$$\epsilon \sim \mathcal{N}(\mu, \sigma^2)$$

where $\mu = 0$ and $\sigma = 0.15$

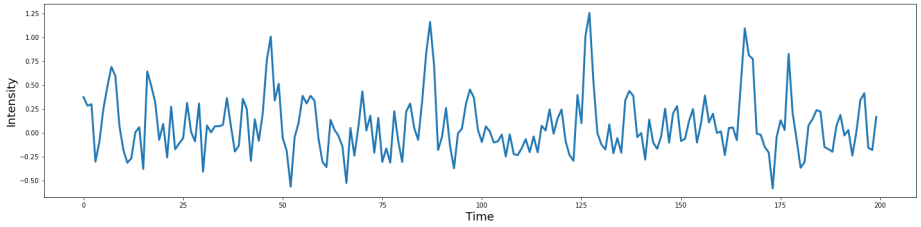
```
sigma = 0.15
epsilon = sigma*np.random.randn(n_tr, 2)
voxel_conv_noise = voxel_conv + epsilon

plot_timeseries(voxel_conv_noise, labels=['Face', 'Object'])
```



Now this is looking much more like real BOLD activity.

Remember, the goal of this exercise is to generate simulated activity from a voxel. If we were to extract signal from a specific voxel we wouldn't know which condition was which, so let's combine these two signals into a single simulated voxel timeseries by adding the two vectors together with the `.sum()` method.

```
Y = voxel_conv_noise.sum(axis=1)

plot_timeseries(Y)
```



Construct Design Matrix

Now that we have our simulated voxel timeseries, let's try and see if we can recover the original signal using a general linear model in the form of:

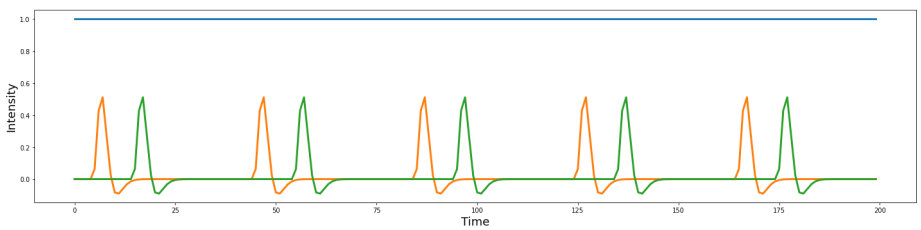
$$Y = X\beta + \epsilon$$

where Y is our observed voxel time series. X is our model or design matrix, and is where we will specify a predicted response to each condition. β is a vector of values that we will estimate to scale our model. ϵ is independent gaussian noise. This model is linear because we can decompose Y into a set of features or independent variables that are scaled by an estimated β parameter and summed together. The ϵ parameter is not usually known and can also be estimated.

You may be wondering how our model is distinct from our simulated data. Remember when we simulated the data we specified 3 parameters - face amplitude, object amplitude, and ϵ, we could have also added a mean, but for now, let's just assume that it is zero. When we fit our model to the simulated data, we should in theory be able to almost perfectly recover these three parameters.

Now let's build a design matrix X using an intercept, and a regressor indicating the onset of each condition, convolved with the hemodynamic response function (HRF).

```
n_tr = 200
n_trial = 5
face = np.zeros(n_tr)
face[np.arange(10, n_tr, int(n_tr/n_trial))] = 1
obj = np.zeros(n_tr)
obj[np.arange(20, n_tr, int(n_tr/n_trial))] = 1
intercept = np.ones(n_tr)
X = np.vstack([intercept, np.convolve(face, hrf, mode='same'), np.convolve(obj, hrf,
mode='same')]).T

plot_timeseries(X)
```



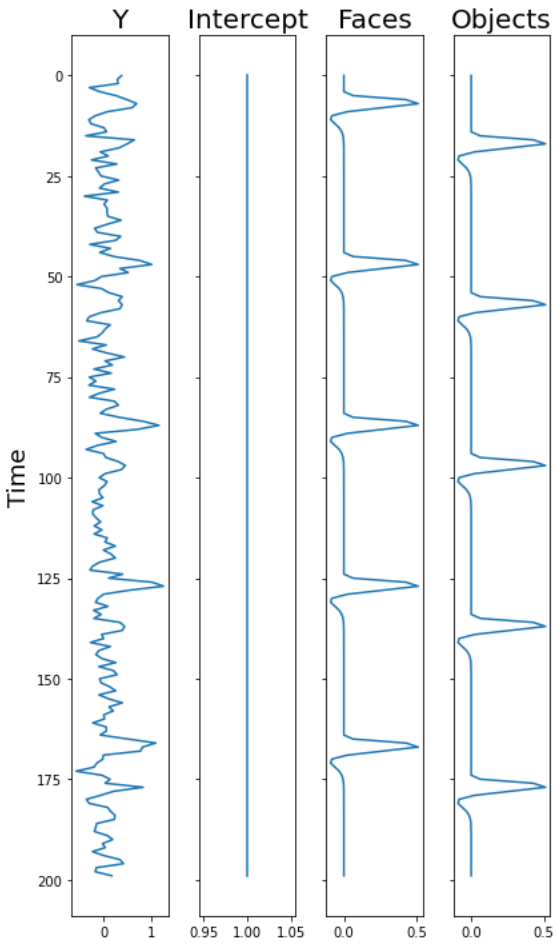
We can write our model out so that it is very clear what we are doing.

$$Voxel = \beta_0 \cdot Intercept + \beta_1 \cdot Faces + \beta_2 \cdot Objects + \epsilon$$

We can also make a plot and rotate the timeseries, to better reflect the equation.

It should be clear how each of these components relate to the regression equation.

```
f, a = plt.subplots(ncols=4, figsize=(6, 10), sharey=True)
a[0].plot(np.expand_dims(Y, axis=1), range(len(Y)))
a[1].plot(X[:,0], range(len(Y)))
a[2].plot(X[:,1], range(len(Y)))
a[3].plot(X[:,2], range(len(Y)))
a[0].set_ylabel('Time', fontsize=18)
a[0].set_title('Y', fontsize=20)
a[1].set_title('Intercept', fontsize=20)
a[2].set_title('Faces', fontsize=20)
a[3].set_title('Objects', fontsize=20)
plt.gca().invert_yaxis()
plt.tight_layout()
```



Estimate GLM

Now that have created our simulated voxel timeseries Y and our design matrix X, we need to fit our model to the data by estimating the three β parameters.

There are several ways to estimate the parameters for our general linear model. The Ordinary Least Squares (OLS) estimator finds the $\hat{\beta}$ hyperplane that minimizes the error between the observed Y and predicted $\hat{Y}$.

This can be formulated using linear algebra as:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

There is also maximum likelihood estimator, which should produce an almost identical result to the ordinary least squares estimator when the error terms are normally distributed.

$$L(\beta, \sigma^2|Y, X) = \prod_{i=1}^{n} \frac{1}{\sqrt{(2\pi\sigma^2)}} \cdot e^{-\frac{(Y_i - \beta X_i)^2}{2\sigma^2}}$$

where

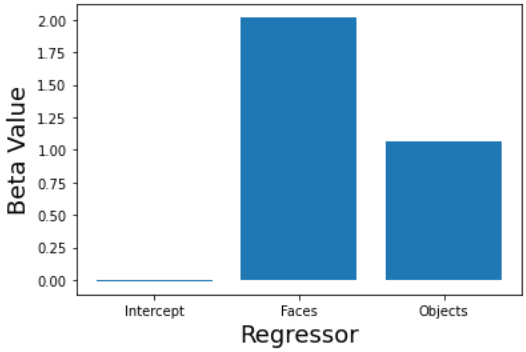
$$\mathcal{N}(0, \sigma^2)$$

For this class, we will primarily be focusing on the Ordinary Least Squares Estimator. In fact, just to demonstrate that the math is actually relatively straightforward, we will write our own function for the estimator using the linear algebra formulation. In practice, we typically will use a premade function, which is usually slightly more computationally efficient and will also calculate standard errors, etc.

For a more in depth overview of GLM estimation, watch this video by Tor Wager and Martin Lindquist.

```
def ols_estimator(X, Y):
    return np.dot(np.dot(np.linalg.pinv(np.dot(X.T, X)), X.T), Y)

beta = ols_estimator(X, Y)

plt.bar(['Intercept','Faces', 'Objects'], beta)
plt.xlabel('Regressor', fontsize=18)
plt.ylabel('Beta Value', fontsize=18)

print(f'beta Faces - beta Objects: {beta[1]-beta[2]:.2}')
```

```
beta Faces - beta Objects: 0.95
```



We can see that our model is working pretty well. We did not add a mean to the simulated timeseries, so our estimator correctly figures out that the intercept parameter should be zero. The model also correctly figured out that the scaling parameter for the faces regressor was 2, and 1 for the objects regressor, with the difference between them equal to approximately 1.

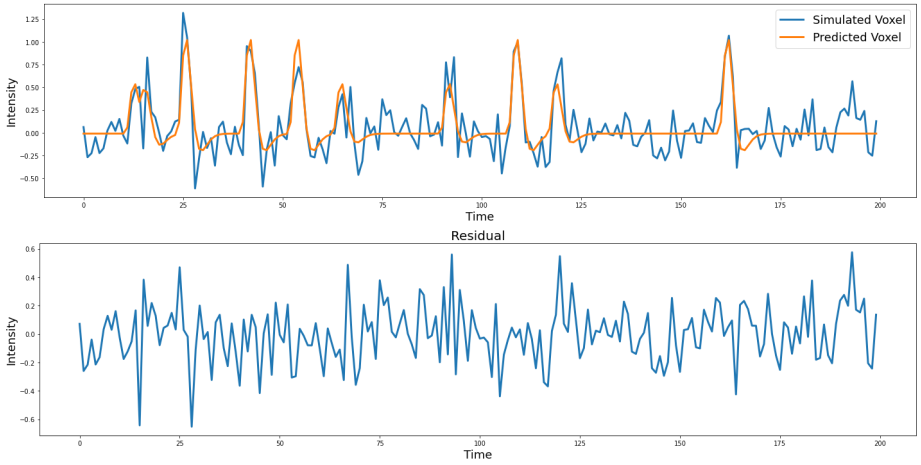
Another way to evaluate how well our model is working is to plot our predicted $\hat{Y}$ on top of our simulated Y.

We can quantify the degree to which our model is accurately predicting the observed data by calculating the residual.

$$residual = Y - \hat{Y}$$

```
predicted_y = np.dot(X, beta)

predicted_ts = np.vstack([Y, predicted_y]).T

plot_timeseries(predicted_ts, labels=['Simulated Voxel', 'Predicted Voxel'])

residual = Y - predicted_y

plot_timeseries(residual)
plt.title('Residual', fontsize=20)
```

```
Text(0.5, 1.0, 'Residual')
```



Standard Errors

As you can see, we are doing a reasonable job recovering the original signals.

You may recall that we specified 3 parameters in our simulation

- a β weight for faces
- a β weight for objects
- an ϵ noise parameter.

The *standard error of the estimate* refers to the standard deviation of the residual.

Formally, this can be described as:

$$\hat{\sigma} = \sqrt{\frac{\sum_{i}^{n}(\hat{Y}_i - Y_i)^2}{n-k}}$$

where n is the number of observations and k is the total number of regressors.

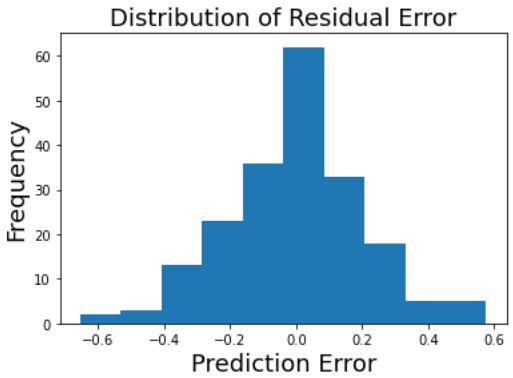
This number is essentially an estimate of the overall amount of error in the model or ϵ. This error is assumed to be independent and normally distributed. The smaller the residual variance $\hat{\sigma}$ the better the fit of the model.

As you can see, the parameter is close, but slightly higher than the one we simulated. This might be because we have relatively little data in our simulation.

```
standard_error_of_estimate = np.std(residual)

print(f"Standard Error of the Estimate: {standard_error_of_estimate:.2}")

plt.hist(residual)
plt.title('Distribution of Residual Error', fontsize=18)
plt.ylabel('Frequency', fontsize=18)
plt.xlabel('Prediction Error', fontsize=18)
```

```
Standard Error of the Estimate: 0.2
```

```
Text(0.5, 0, 'Prediction Error')
```



Explained Variance

Sometimes we want a single metric to quantify overall how well our model was able to explain variance in the data. There are many metrics that can provide a quantitative measure of *goodness of fit*.

Here we will calculate R^2 using the following formula:

$$R^2 = 1 - \frac{\sum_i^n (\hat{y}_i - y_i)^2}{\sum_i^n (y_i - \bar{y})^2}$$

where y_i is the measured value of the voxel at timepoint i, $\hat{y}_i$ is the predicted value for time point i, and $\bar{y}$ is the mean of the measured voxel timeseries.

R^2 will lie on the interval between $[0, 1]$ and can be interpreted as percentage of the total variance in Y explained by the model, X, where 1 is 100% and 0 is none.

```
def r_square(Y, predicted_y):
    SS_total = np.sum((Y - np.mean(Y))**2)
    SS_residual = np.sum((Y - predicted_y)**2)
    return 1-(SS_residual/SS_total)

print(f"R^2: {r_square(Y, predicted_y):.2}")
```

```
R^2: 0.6
```

Standard Error of β estimates

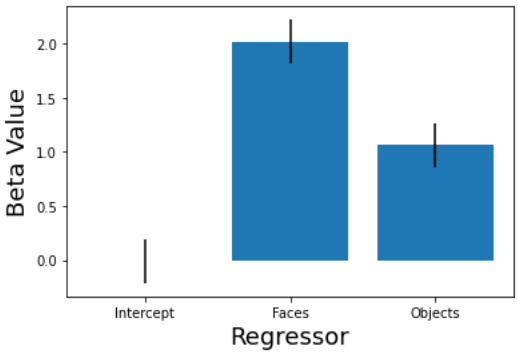
We can also estimate the uncertainty of regression coefficients. The uncertainty of the beta parameters is quantified as a standard error around each specific estimate.

$$\sigma = \sqrt{diag((X^T X)^{-1})} \cdot \hat{\sigma}$$

This is essentially a confidence interval around the β_j estimate. One standard error, $1 * \hat{\sigma}$ is approximately equivalent to a 68% confidence interval, while $2 * \hat{\sigma}$ is approximately a 95% confidence interval.

```
std_error = np.sqrt(np.diag((np.linalg.pinv(np.dot(X.T, X))))) *
standard_error_of_estimate

plt.bar(['Intercept','Faces', 'Objects'], beta, yerr = standard_error_of_estimate)
plt.xlabel('Regressor', fontsize=18)
plt.ylabel('Beta Value', fontsize=18)
```

```
Text(0, 0.5, 'Beta Value')
```



Statistical Significance

We could also perform a hypothesis test to evaluate if any of the regressors are statistically different from zero.

This exercise is simply meant to provide parallels to common statistical jargon. In practice, this is actually rarely done in neuroimaging analysis as we are typically more interested in making statistical inferences across the population rather than within a single participant.

The formula for calculating a t-statistic is very simple:

$$t = \frac{\hat{\beta}_j}{\hat{\sigma}_j}$$

where β_j refers to the estimated parameter for a regressor j, and σ_j refers to the standard error of regressor j.

t values that are more than 2 standard errors away from zero are called *statistically significant*, which basically just means we are more confident that the estimate is stable and not just an artifact of small sample size. In general, we don't recommend reading too much into significance for individual β estimates in single subject fMRI analysis.

```
t = beta/std_error
t
```

```
array([-0.65716867, 15.83929013,  8.23580115])
```

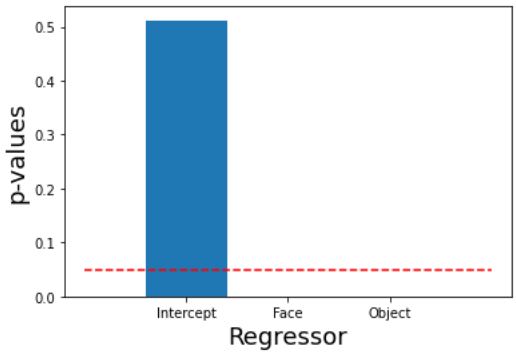
Just like in intro statistics, we could find the p-value that corresponds to a particular t-statistic using the t-distribution. We will load the t distribution from `scipy.stats` and calculate the corresponding p-values using the survival function or $1 - cdf$, which requires specifying the degrees of freedom (df), which is $n - 1$. We multiply these values by 2 to calculated a two-tailed test.

You can see that the intercept β is not significant, but the face and object regressors are well below `p < 0.05`.

```
from scipy import stats

p = stats.t.sf(np.abs(t), n_tr-1)*2
print(p)

plt.bar(['Intercept', 'Face', 'Object'], p)
plt.ylabel('p-values', fontsize=18)
plt.xlabel('Regressor', fontsize=18)
plt.hlines(0.05, -1, 3, linestyles='dashed', color='red')
```

```
[5.11831798e-01 4.26134329e-37 2.33906074e-14]
```

```
<matplotlib.collections.LineCollection at 0x7fe020f135d0>
```

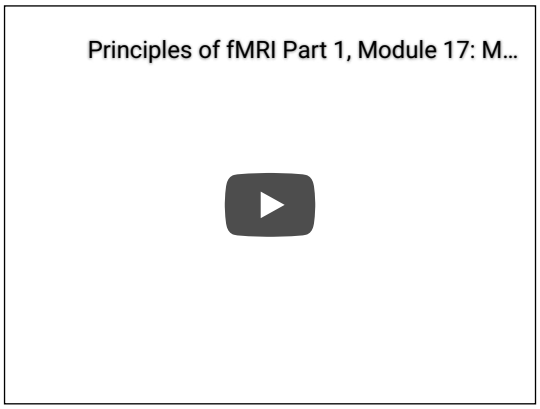


Contrasts

Contrasts are a very important concept in fMRI data analysis as they provide the statistical inference underlying the subtraction method of making inferences.

Let's watch a short video by Tor Wager on contrasts. We will also spend much more time on contrasts in the group analysis tutorial. We also recommend watching Jeannette Mumford's overview of contrasts for a more statistical perspective.

```
YouTubeVideo('7MibM1ATai4')
```



Contrasts describe a linear combination of variables in a regression model whose coefficients add up to zero. This allows us to flexibly compare different experimental conditions.

For example, suppose we just wanted to know the magnitude of an effect for a single condition, such as the brain response to faces. We would create a contrast code that isolates the effect size (i.e., β estimate for the face regressor)

If our GLM, was:

$$Y = \beta_0 \cdot Intercept + \beta_1 \cdot Faces + \beta_2 \cdot Objects$$

then, the corresponding contrast code or vector for faces would be:

[0, 1, 0]

The contrast code for the object condition would be:

[0, 0, 1]

and importantly the contrast *between* the face and object condition would be:

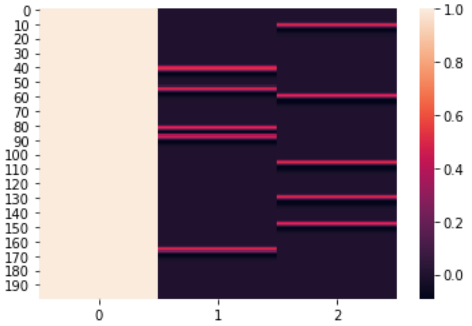
[0, 1, -1]

More simply, we are calculating the magnitude of the effect of the difference between viewing faces and objects in a single voxel.

To make this a little bit more clear, we will show a graphical representation of the design matrix to make it obvious what we are contrasting.

```
sns.heatmap(X)

c1 = [0, 1, 0]
c2 = [0, 0, 1]
c3 = [0, 1, -1]

{(str(c),np.dot(beta, c)) for c in [c1, c2, c3]}
```

```
{('[0, 0, 1]', 1.0721521304389894),
 ('[0, 1, -1]', 0.8735019759362619),
 ('[0, 1, 0]', 1.9456541063752513)}
```



Efficiency

We can estimate the efficiency, or the quality of an estimator for a specific experimental design or a hypothesis testing procedure. Efficiency is related to power, or the ability to detect an effect should one exist. However, unlike power, we can estimate efficiency from our design matrix and do not actually need to know the standard error for the model (unlike with power calculations). Specifically, efficiency is defined as the inverse of the sum of the estimator variances. For a more detailed explanation and general experimental design recommendations see this overview by Rik Henson, or this blog post on efficiency in experimental designs.

$$e(c\hat{\beta}) = \frac{1}{c(X^T X)^{-1} c^T}$$

Reducing collinearity or covariance between regressors can increase design efficiency

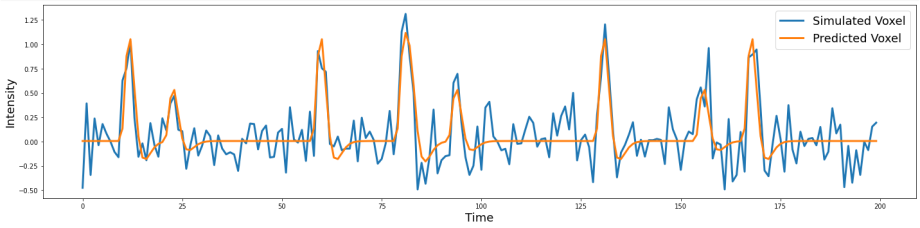
```
def contrast_efficiency(X, contrast):
    c = np.array(contrast)
    return 1/(np.dot(np.dot(c, np.linalg.pinv(np.dot(X.T, X))), c.T))

c1 = [0, 1, 0]
c2 = [0, 0, 1]
c3 = [0, 1, -1]

[contrast_efficiency(X, x) for x in [c1, c2, c3]]
```

```
[2.44032677542221, 2.5429401030547045, 1.3578767952519295]
```

Varying the Inter-Trial Interval with Jittering

```
# Set Simulation Parameters
n_tr = 200
n_trial = 5
face_intensity = 2
object_intensity = 1
sigma = 0.15

# Build Simulation
face_trials = np.random.randint(10, 180, n_trial)
obj_trials = np.random.randint(10, 180, n_trial)
face = np.zeros(n_tr)
face[face_trials] = 1
obj = np.zeros(n_tr)
obj[obj_trials] = 1
voxel_conv = np.vstack([np.convolve(face*face_intensity, hrf, mode='same'),
np.convolve(obj*object_intensity, hrf, mode='same')]).T
epsilon = sigma*np.random.randn(n_tr, 2)
voxel_conv_noise = voxel_conv + epsilon

# Build Model
Y = voxel_conv_noise.sum(axis=1)
X = np.vstack([np.ones(len(face)), np.convolve(face, hrf, mode='same'),
np.convolve(obj, hrf, mode='same')]).T

# Estimate Model
beta = ols_estimator(X, Y)
predicted_y = np.dot(X, beta)
predicted_sigma = np.std(residual)
predicted_ts = np.vstack([Y, predicted_y]).T
plot_timeseries(predicted_ts, labels=['Simulated Voxel', 'Predicted Voxel'])

# Estimate Contrast Efficiency
[contrast_efficiency(X, x) for x in [c1, c2, c3]]
```

```
[2.5597167760964403, 2.066820546582558, 1.1233445684652128]
```



Autocorrelation

The BOLD signal has some intrinsic autocorrelation that varies with the length of the TR. Different software packages have provided varying solutions to this problem. For example, SPM implements an AR(1) model, which means that it trys to account for the fact that the signal is consistently correlated (i.e., autoregressive) with one lag. In practice, these will rarely change the beta estimates, but rather will adjust our standard errors around the estimates. As we will discuss soon, most group level analyses ignore these subject level, or first-level errors anyway. It is debatable if this is actually a good practice, but it reduces the importance of accounting for autocorrelation when looking at group level statistics in standard experimental design.

Another important thing to note is that there is some evidence that the AR(1) model can actually increase false positives, especiall in shorter TRs. See this paper by Anders Ekland and colleagues for more details. Also, this is a helpful blog post discussing prewhitening.

For the scope of this course we will largely be ignoring this issue, but I will plan to add some examples and simulations in the future. For now, I encourage you to watch this video on AR models if you are interested in learning more about this topic.

Exercises

For our homework exercises, let's use the simulation to answer questions we might have about experimental design.

Exercise 1. What happens when we vary the signal amplitude?

Some signals will be very strong and others weaker. How does the model fit change when the signal amplitudes are stronger and weaker?

In this exercise, make a plot showing how the r^2 changes over 3 different levels of signal intensity.

Exercise 2. What happens when we vary the noise?

How does the amount of noise in the data impact our model fits?

In this exercise, make a plot showing the r^2 for 3 different levels of simulated noise.

Exercise 3. How many trials do we need?

A common question in experimental design is determining the optimal number of trials.

In this exercise, try evaluating how 3 different numbers of trials might impact the contrast efficiency.

Exercise 4. What is the impact of the stimulus duration?

What if one condition simply results in processes that systematically take longer than the other condition?

In this exercise, let's try to answer this question by creating a simulation where the signal intensity between the two condition is identical, but one simply has a longer duration (i.e., the duration has more TRs than the other condition).

Make a plot showing what happens to the β estimates.

Modeling Single Subject Data

Written by Luke Chang

Now that we have learned the basics of the GLM using simulations, it's time to apply this to working with real data. The first step in fMRI data analysis is to build a model for each subject to predict the activation in a single voxel over the entire scanning session. To do this, we need to build a design matrix for our general linear model. We expect distinct brain regions to be involved in processing specific aspects of our task. This means that we will construct separate regressors that model different brain processes.

In this tutorial, we will learn how to build and estimate a single subject first-level model and will cover the following topics:

- Building a design matrix
- Modeling noise in the GLM with nuisance variables
- Estimating GLM
- Performing basic contrasts

Dataset

We will continue to work with the Pinel Localizer dataset from our preprocessing examples.

The Pinel Localizer task was designed to probe several different types of basic cognitive processes, such as visual perception, finger tapping, language, and math. Several of the tasks are cued by reading text on the screen (i.e., visual modality) and also by hearing auditory instructions (i.e., auditory modality). The trials are randomized across conditions and have been optimized to maximize efficiency for a rapid event related design. There are 100 trials in total over a 5-minute scanning session. Read the original paper for more specific details about the task and the dataset paper.

This dataset is well suited for these tutorials as it is (a) publicly available to anyone in the world, (b) relatively small (only about 5min), and (c) provides many options to create different types of contrasts.

There are a total of 94 subjects available, but we will primarily only be working with a smaller subset of 10-20 participants. See our tutorial on how to download the data if you are not taking the Psych60 version of the class.

Building a Design Matrix

First, we will learn the basics of how to build a design matrix for our GLM.

Let's load all of the python modules we will need to complete this tutorial.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import nibabel as nib
from nltools.file_reader import onsets_to_dm
from nltools.stats import regress, zscore
from nltools.data import Brain_Data, Design_Matrix
from nltools.stats import find_spikes
from nilearn.plotting import view_img, glass_brain, plot_stat_map
from bids import BIDSLayout, BIDSValidator

data_dir = '../data/localizer'
layout = BIDSLayout(data_dir, derivatives=True)
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-
packages/sklearn/utils/deprecation.py:144: FutureWarning: The
sklearn.linear_model.base module is  deprecated in version 0.22 and will be removed
in version 0.24. The corresponding classes / functions should instead be imported
from sklearn.linear_model. Anything that cannot be imported from sklearn.linear_model
is now part of the private API.
  warnings.warn(message, FutureWarning)
```

To build the design matrix, we will be using the Design_Matrix class from the nltools toolbox. First, we use pandas to load the text file that contains the onset and duration for each condition of the task. Rows reflect measurements in time sampled at 1/tr cycles per second. Columns reflect distinct conditions. Conditions are either on or off. We then cast this Pandas DataFrame as a Design_Matrix object. Be sure to specify the sampling frequency, which is $\frac{1}{tr}$.

```
def load_bids_events(layout, subject):
    '''Create a design_matrix instance from BIDS event file'''

    tr = layout.get_tr()
    n_tr = nib.load(layout.get(subject=subject, scope='raw', suffix='bold')
[0].path).shape[-1]

    onsets = pd.read_csv(layout.get(subject=subject, suffix='events')[0].path,
sep='\t')
    onsets.columns = ['Onset', 'Duration', 'Stim']
    return onsets_to_dm(onsets, sampling_freq=1/tr, run_length=n_tr)

dm = load_bids_events(layout, 'S01')
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/nltools-0.3.18-
py3.7.egg/nltools/file_reader.py:141: UserWarning: Computed onsets for
video_right_hand are inconsistent with expected values. Please manually verify the
outputted Design_Matrix!
  f"Computed onsets for {data.Stim.unique()[i]} are inconsistent with expected
values. Please manually verify the outputted Design_Matrix!"
/Users/lukechang/anaconda3/lib/python3.7/site-packages/nltools-0.3.18-
py3.7.egg/nltools/file_reader.py:141: UserWarning: Computed onsets for
video_left_hand are inconsistent with expected values. Please manually verify the
outputted Design_Matrix!
  f"Computed onsets for {data.Stim.unique()[i]} are inconsistent with expected
values. Please manually verify the outputted Design_Matrix!"
/Users/lukechang/anaconda3/lib/python3.7/site-packages/nltools-0.3.18-
py3.7.egg/nltools/file_reader.py:141: UserWarning: Computed onsets for
audio_computation are inconsistent with expected values. Please manually verify the
outputted Design_Matrix!
  f"Computed onsets for {data.Stim.unique()[i]} are inconsistent with expected
values. Please manually verify the outputted Design_Matrix!"
```

The Design_Matrix class is built on top of Pandas DataFrames and retains most of that functionality. There are additional methods to help with building design matrices. Be sure to check out this tutorial for more information about how to use this tool.

We can check out details about the data using the `.info()` method.

```
dm.info()
```

```
<class 'nltools.data.design_matrix.Design_Matrix'>
RangeIndex: 128 entries, 0 to 127
Data columns (total 10 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   video_computation       128 non-null    float64
 1   horizontal_checkerboard  128 non-null    float64
 2   audio_right_hand        128 non-null    float64
 3   audio_sentence          128 non-null    float64
 4   video_right_hand        128 non-null    float64
 5   audio_left_hand         128 non-null    float64
 6   video_left_hand         128 non-null    float64
 7   vertical_checkerboard   128 non-null    float64
 8   audio_computation       128 non-null    float64
 9   video_sentence          128 non-null    float64
dtypes: float64(10)
memory usage: 10.1 KB
```

We can also view the raw design matrix as a dataframe just like pd.Dataframe. We use the `.head()` method to just post the first few rows.

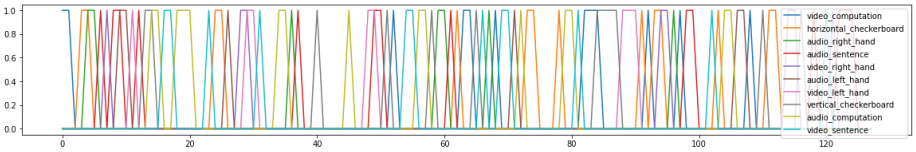
```
dm.head()
```

	video_computation	horizontal_checkerboard	audio_right_hand	audio_sentence	video_right_ha
0	1.0	0.0	0.0	0.0	
1	1.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	
4	0.0	1.0	1.0	0.0	

We can plot each regressor's time course using the `.plot()` method.

```
f,a = plt.subplots(figsize=(20,3))
dm.plot(ax=a)
```

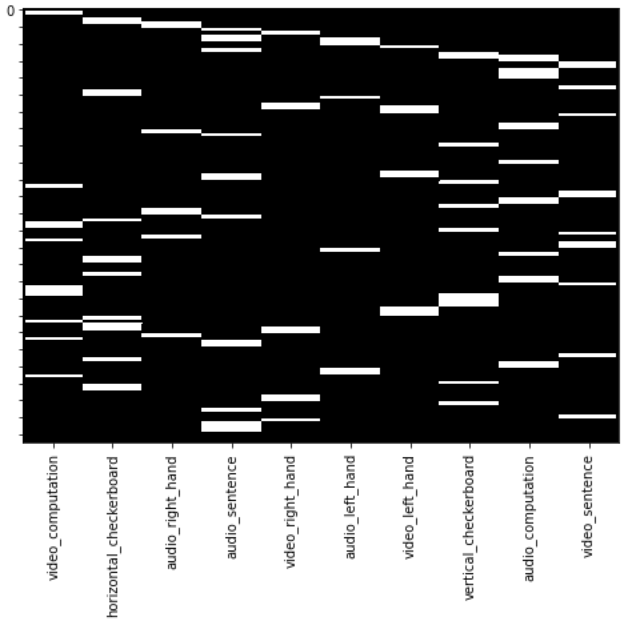
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb700ed4110>
```



This plot can be useful sometimes, but here there are too many regressors, which makes it difficult to see what is going on.

Often, `.heatmap()` method provides a more useful visual representation of the design matrix.

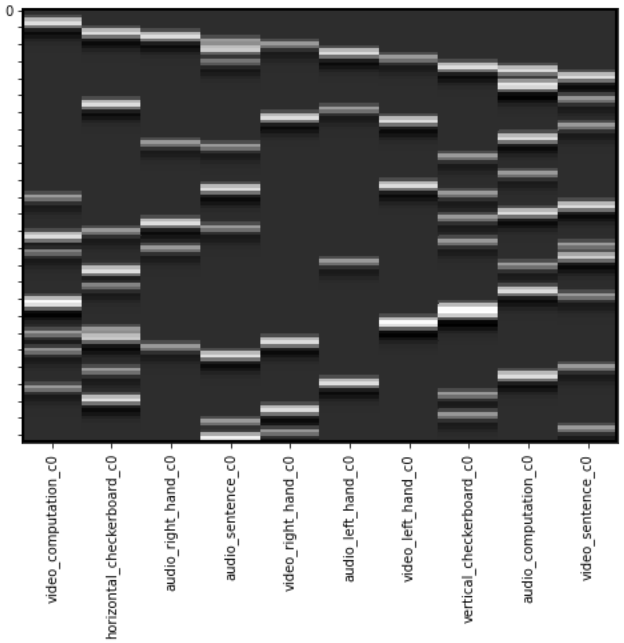
```
dm.heatmap()
```



HRF Convolution

Recall what we learned about convolution in our signal processing tutorial. We can now convolve all of the onset regressors with an HRF function using the `.convolve()` method. By default it will convolve all regressors with the standard double gamma HRF function, though you can specify custom ones and also specific regressors to convolve. Check out the docstrings for more information by adding a `?` after the function name. If you are interested in learning more about different ways to model the HRF using temporal basis functions, watch this video.

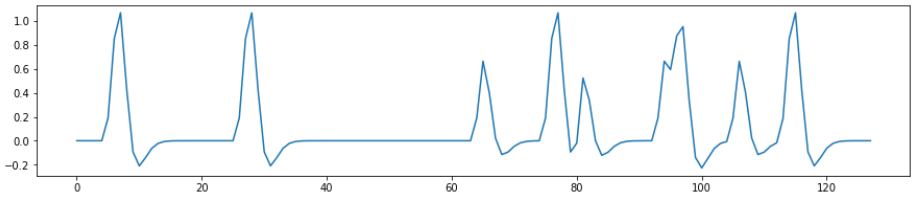
```
dm_conv = dm.convolve()
dm_conv.heatmap()
```



You can see that each of the regressors is now bit blurrier and now has the shape of an HRF function. We can plot a single regoressor to see this more clearly using the `.plot()` method.

```
f,a = plt.subplots(figsize=(15,3))
dm_conv['horizontal_checkerboard_c0'].plot(ax=a)
```

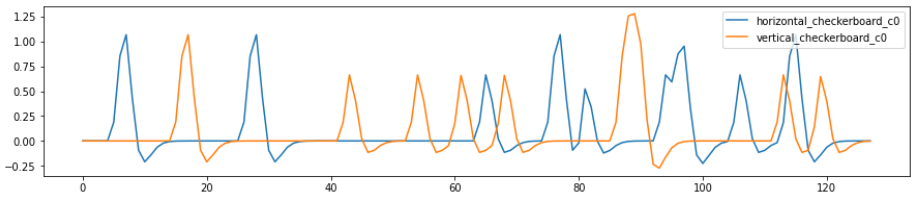
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb69091c990>
```



Maybe we want to plot both of the checkerboard regressors.

```
f,a = plt.subplots(figsize=(15,3))
dm_conv[['horizontal_checkerboard_c0','vertical_checkerboard_c0']].plot(ax=a)
```

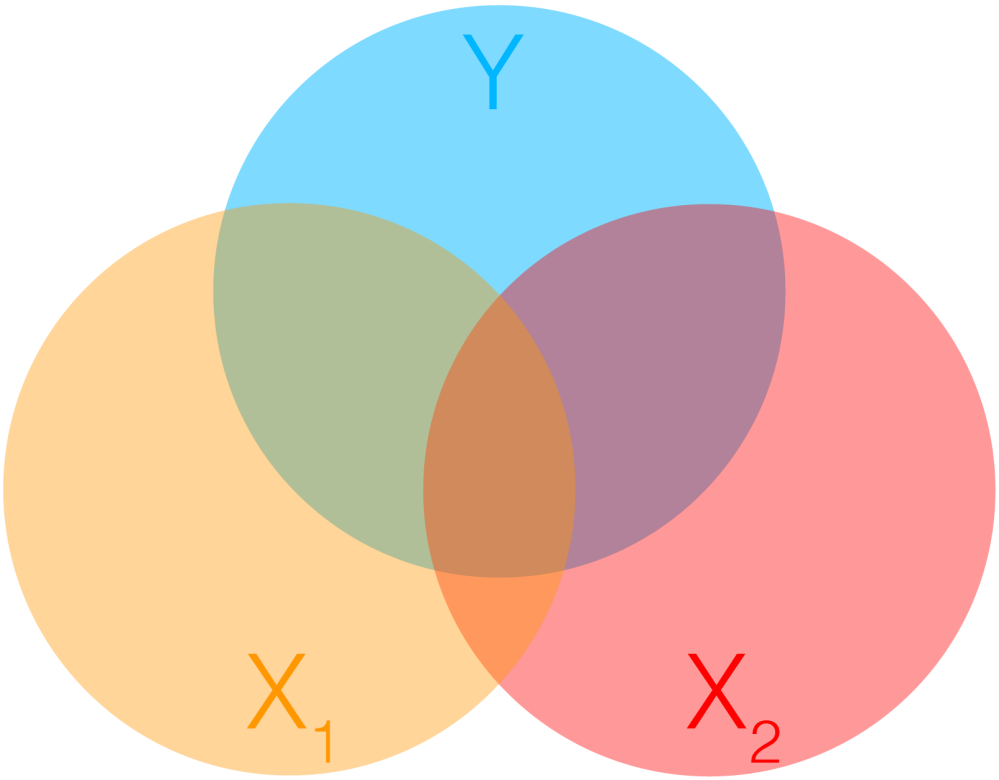
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb690964c90>
```



Multicollinearity

In statistics, collinearity or multicollinearity is when one regressor can be strongly linearly predicted from the others. While this does not actually impact the model's ability to predict data as a whole, it will impact our ability to accurately attribute variance to a single regressor. Recall that in multiple regression, we are estimating the independent variance from each regressor from X on Y. If there is substantial overlap between the regressors, then the estimator can not attribute the correct amount of variance each regressor accounts for Y and the coefficients can become unstable. A more intuitive depiction of this problem can be seen in the venn diagram. The dark orange area in the center at the confluence of all 3 circles reflects the shared variance between X1 and X2 on Y. If this area becomes bigger, the unique variances become smaller and individually reflect less of the total variance on Y.

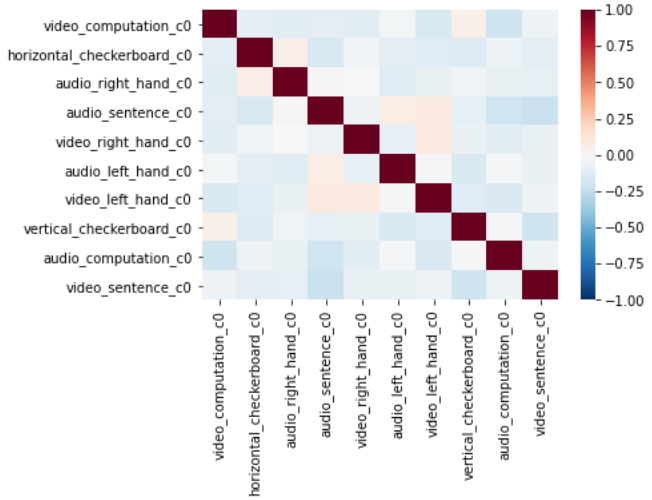
$$Y = b_0 + b_1 * X_1 + b_2 * X_2$$



One way to evaluate multicollinearity is to examine the pairwise correlations between each regressor. We plot the correlation matrix as a heatmap.

```
sns.heatmap(dm_conv.corr(), vmin=-1, vmax=1, cmap='RdBu_r')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb700e335d0>
```



Variance Inflation Factor

Pairwise correlations will let you know if any regressor is correlated with another regressor. However, we are even more concerned about being able to explain any regressor as a linear combination of the other regressors. For example, *can one regressor be explained by three or more of the remaining regressors?* The variance

inflation factor (VIF) is a metric that can help us detect multicollinearity. Specifically, it is simply the ratio of variance in a model with multiple terms, divided by the variance of a model with only a single term. This ratio reduces to the following formula:

$$VIF_j = \frac{1}{1 - R_i^2}$$

Where R_j^2 is the R^2 value obtained by regressing the jth predictor on the remaining predictors. This means that each regressor j will have it's own variance inflation factor.

How should we interpret the VIF values?

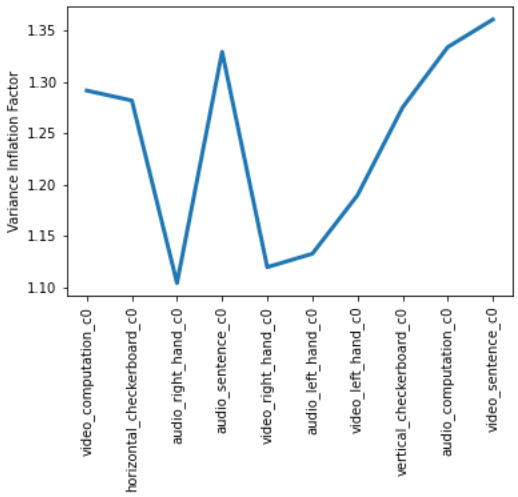
A VIF of 1 indicates that there is no correlation among the jth predictor and the remaining variables. Values greater than 4 should be investigated further, while VIFs exceeding 10 indicate significant multicollinearity and will likely require intervention.

Here we will use the `.vif()` method to calculate the variance inflation factor for our design matrix.

See this overview for more details on VIFs.

```
plt.plot(dm_conv.columns, dm_conv.vif(), linewidth=3)
plt.xticks(rotation=90)
plt.ylabel('Variance Inflation Factor')
```

```
Text(0, 0.5, 'Variance Inflation Factor')
```

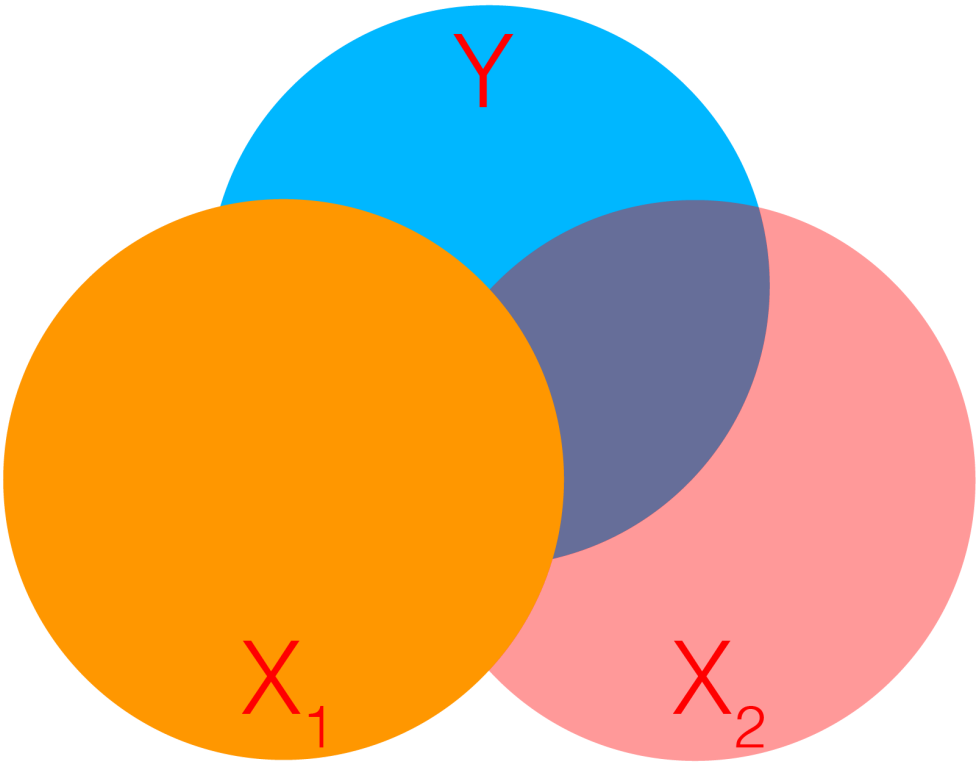


Orthogonalization

There are many ways to deal with collinearity. In practice, don't worry about collinearity between your covariates. The more pernicious issues are collinearity in your experimental design.

It is commonly thought that using a procedure called orthogonalization should be used to address issues of multicollinearity. In linear algebra, orthogonalization is the process of prioritizing shared variance between regressors to a single regressor. Recall that the standard GLM already accounts for shared variance by removing it from individual regressors. Orthogonalization allows a user to assign that variance to a specific regressor. However, the process of performing this procedure can introduce artifact into the model and often changes the interpretation of the beta weights in unanticipated ways.

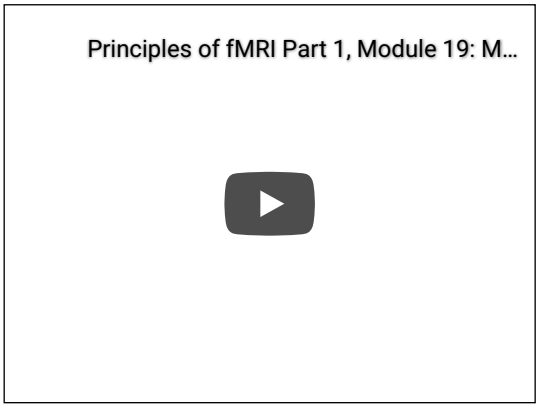
$$Y = b_0 + b_1*X_1 + b_2*X_2$$



In general, we do not recommend using orthogonalization in most use cases, with the exception of centering regressor variables. We encourage the interested reader to review this very useful overview of collinearity and orthogonalization by Jeanette Mumford and colleagues.

Nuisance Variables

```
from IPython.display import YouTubeVideo

YouTubeVideo('DEtwsFdFwYc')
```



Filtering

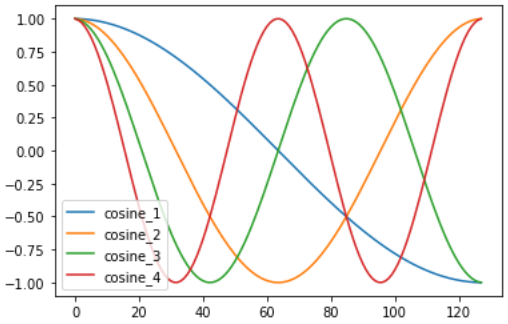
Recall from our signal processing tutorial, that there are often other types of artifacts in our signal that might take the form of slow or fast oscillations. It is common to apply a high pass filter to the data to remove low frequency artifacts. Often this can also be addressed by simply using a few polynomials to model these types of trends. If we were to directly filter the brain data using something like a butterworth filter as we did in our signal processing

tutorial, we would also need to apply it to our design matrix to make sure that we don't have any low frequency drift in experimental design. One easy way to simultaneously perform both of these procedures is to simply build a filter into the design matrix. We will be using a discrete cosine transform (DCT), which is a basis set of cosine regressors of varying frequencies up to a filter cutoff of a specified number of seconds. Many software use 100s or 128s as a default cutoff, but we encourage caution that the filter cutoff isn't too short for your specific experimental design. Longer trials will require longer filter cutoffs. See this paper for a more technical treatment of using the DCT as a high pass filter in fMRI data analysis. In addition, here is a more detailed discussion about filtering.

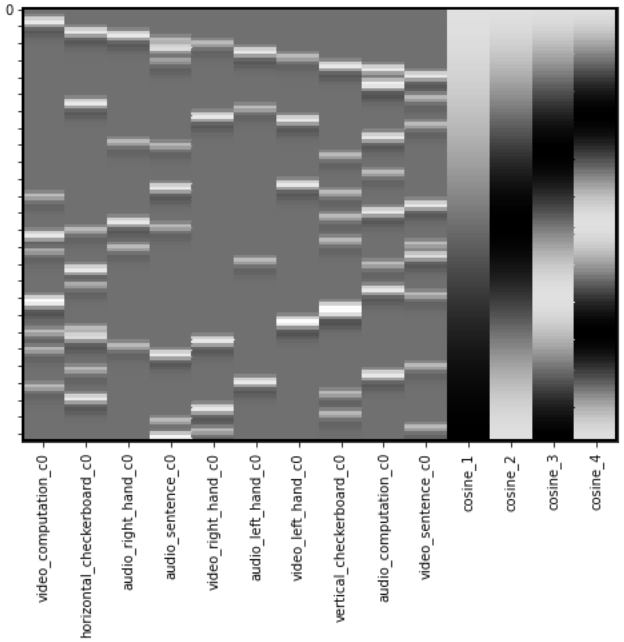
```
dm_conv_filt = dm_conv.add_dct_basis(duration=128)
```

```
dm_conv_filt.iloc[:,10:].plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb69095e390>
```




```
dm_conv_filt = dm_conv.add_dct_basis(duration=128)
dm_conv_filt.heatmap()
```

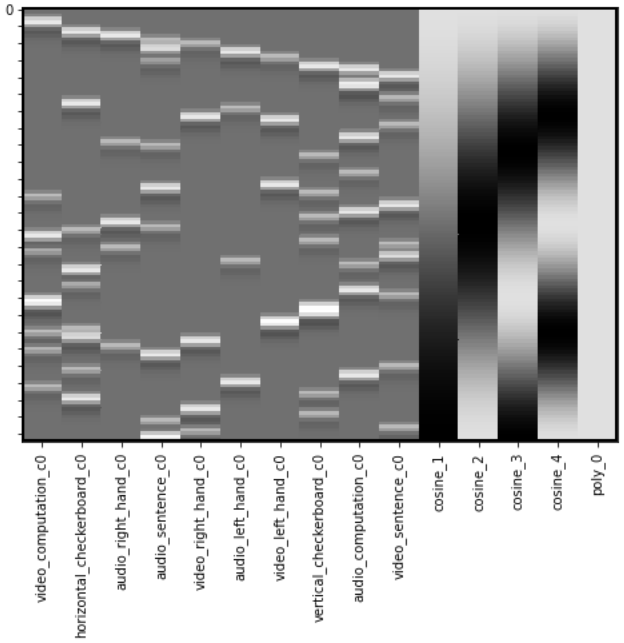


Intercepts

We almost always want to include an intercept in our model. This will usually reflect the baseline, or the average voxel response during the times that are not being modeled as a regressor. It is important to note that you must have some sparsity to your model, meaning that you can't model every point in time, as this will make your model rank deficient and unestimable.

If you are concatenating runs and modeling them all together, it is recommended to include a separate intercept for each run, but not for the entire model. This means that the average response within a voxel might differ across runs. You can add an intercept by simply creating a new column of ones (e.g., `dm['Intercept] = 1`). Here we provide an example using the `.add_poly()` method, which adds an intercept by default.

```
dm_conv_filt_poly = dm_conv_filt.add_poly()
dm_conv_filt_poly.heatmap()
```

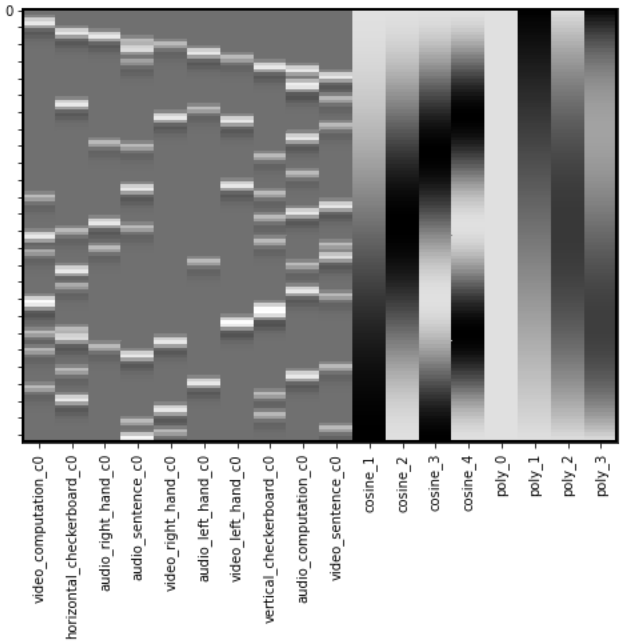


Linear Trends

We also often want to remove any slow drifts in our data. This might include a linear trend and a quadratic trend. We can also do this with the `.add_poly()` method and adding all trends up to an order of 2 (e.g., quadratic). We typically use this approach rather than applying a high pass filter when working with naturalistic viewing data.

Notice that these do not appear to be very different from the high pass filter basis set. It's actually okay if there is collinearity in our covariate regressors. Collinearity is only a problem when it correlates with the task regressors as it means that we will not be able to uniquely model the variance. The DCT can occasionally run into edge artifacts, which can be addressed by the linear trend.

```
dm_conv_filt_poly = dm_conv_filt.add_poly(order=3, include_lower=True)
dm_conv_filt_poly.heatmap()
```



Noise Covariates

Another important thing to consider is removing variance associated with head motion. Remember the preprocessed data has already realigned each TR in space, but head motion itself can nonlinearly distort the magnetic field. There are several common strategies for trying to remove artifacts associated with head motion. One is using a data driven denoising algorithm like ICA and combining it with a classifer such as FSL's FIX module. Another approach is to include the amount of correction that needed to be applied to align each TR. For example, if someone moved a lot in a single TR, there will be a strong change in their realignment parameters. It is common to include the 6 parameters as covariates in your regression model. However, as we already noted, often motion can have a nonlinear relationship with signal intensity, so it is often good to include other transformations of these signals to capture nonlinear signal changes resulting from head motion. We typically center the six realigment parameters (or zscore) and then additionally add a quadratic version, a derivative, and the square of the derivatives, which becomes 24 additional regressors.

In addition, it is common to model out big changes using a regressor with a single value indicating the timepoint of the movement. This will be zeros along time, with a single value of one at the time point of interest. This effectively removes any variance associated with this single time point. It is important to model each "spike" as a separate regressor as there might be distinct spatial patterns associated with different types of head motions. We strongly recommond against using a single continuous frame displacement metric as is often recommended by the fMRIprep team. This assumes (1) that there is a *linear* relationship between displacement and voxel activity, and (2) that there is a *single* spatial generator or pattern associated with frame displacement. As we saw in the ICA noise lab, there might be many different types of head motion artifacts. This procedure of including spikes as nuisance regressors is mathematically equivalent to censoring your data and removing the bad TRs. We think it is important to do this in the context of the GLM as it will also reduce the impact if it happens to covary with your task.

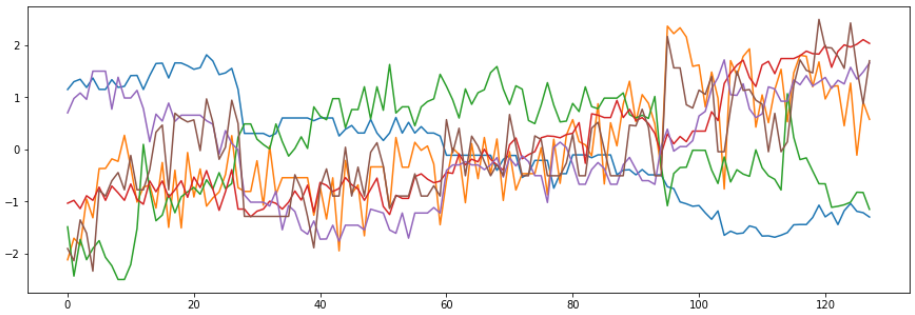
First, let's load preprocessed data from one participant.

```
sub = 'S01'
data = Brain_Data(layout.get(subject=sub, task='localizer', scope='derivatives',
suffix='bold', extension='nii.gz', return_type='file')[1])
```

Now let's inspect the realignment parameters for this participant. These pertain to how much each volume had to be moved in the (X,Y,Z) planes and rotations around each axis. We are standardizing the data so that rotations and translations are on the same scale.

```
covariates = pd.read_csv(layout.get(subject='S01', scope='derivatives',
extension='.tsv')[0].path, sep='\t')

mc = covariates[['trans_x','trans_y','trans_z','rot_x', 'rot_y', 'rot_z']]

plt.figure(figsize=(15,5))
plt.plot(zscore(mc))
```

```
[<matplotlib.lines.Line2D at 0x7fb68b61c190>,
 <matplotlib.lines.Line2D at 0x7fb68b61cd10>,
 <matplotlib.lines.Line2D at 0x7fb68b61c390>,
 <matplotlib.lines.Line2D at 0x7fb68b61c110>,
 <matplotlib.lines.Line2D at 0x7fb68b61cc50>,
 <matplotlib.lines.Line2D at 0x7fb68b61c290>]
```

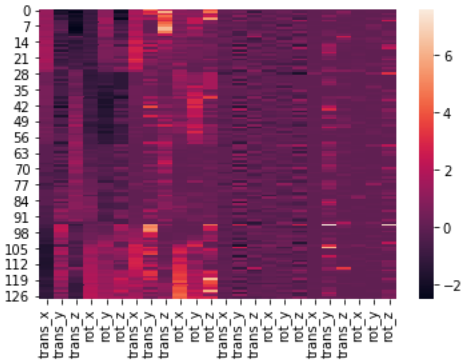


Now, let's build the 24 covariates related to head motion. We include the 6 realignment parameters that have been standardized. In addition, we add their quadratic, their derivative, and the square of their derivative.

We can create a quick visualization to see what the overall pattern is across the different regressors.

```
def make_motion_covariates(mc, tr):
    z_mc = zscore(mc)
    all_mc = pd.concat([z_mc, z_mc**2, z_mc.diff(), z_mc.diff()**2], axis=1)
    all_mc.fillna(value=0, inplace=True)
    return Design_Matrix(all_mc, sampling_freq=1/tr)

tr = layout.get_tr()
mc_cov = make_motion_covariates(mc, tr)

sns.heatmap(mc_cov)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb68bb6d210>
```

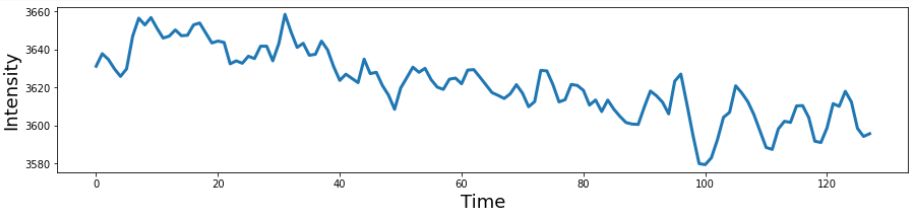


Now let's try to find some spikes in the data. This is performed by finding TRs that exceed a global mean threshold and also that exceed an overall average intensity change by a threshold. We are using an arbitrary cutoff of 3 standard deviations as a threshold.

First, let's plot the average signal intensity across all voxels over time.

```
plt.figure(figsize=(15,3))
plt.plot(np.mean(data.data, axis=1), linewidth=3)
plt.xlabel('Time', fontsize=18)
plt.ylabel('Intensity', fontsize=18)
```

```
Text(0, 0.5, 'Intensity')
```

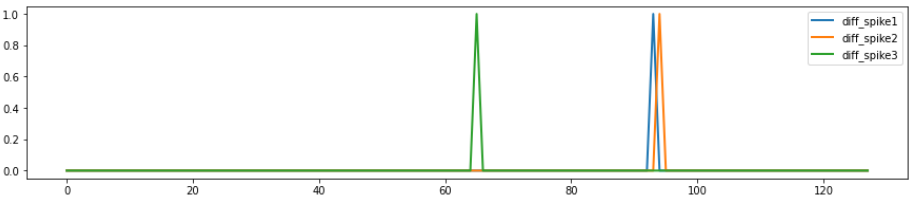


Notice there is a clear slow drift in the signal that we will need to remove with our high pass filter.

Now, let's see if there are any spikes in the data that exceed our threshold. What happens if we use a different threshold?

```
spikes = data.find_spikes(global_spike_cutoff=2.5, diff_spike_cutoff=2.5)

f, a = plt.subplots(figsize=(15,3))
spikes = Design_Matrix(spikes.iloc[:,1:], sampling_freq=1/tr)
spikes.plot(ax = a, linewidth=2)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb7010e7950>
```

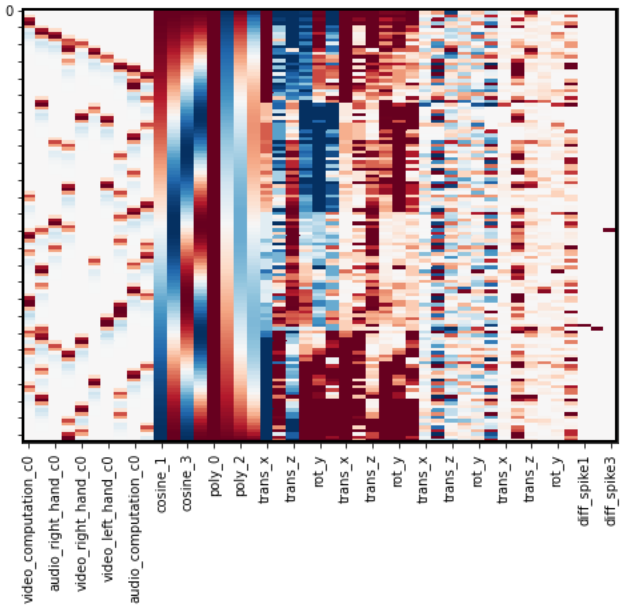


For this subject, our spike identification procedure only found a single spike. Let's add all of these covariate to our design matrix.

In this example, we will append each of these additional matrices to our main design matrix.

Note: `.append()` requires that all matrices are a design_matrix with the same sampling frequency.

```
dm_conv_filt_poly_cov = pd.concat([dm_conv_filt_poly, mc_cov, spikes], axis=1)
dm_conv_filt_poly_cov.heatmap(cmap='RdBu_r', vmin=-1,vmax=1)
```



Smoothing

To increase the signal to noise ratio and clean up the data, it is common to apply spatial smoothing to the image.

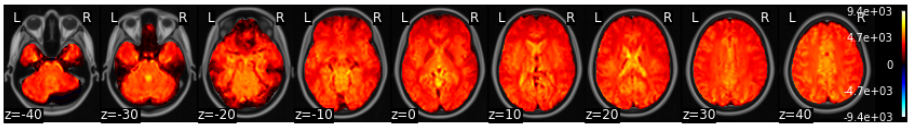
Here we will convolve the image with a 3-D gaussian kernel, with a 6mm full width half maximum (FWHM) using the `.smooth()` method.

```
fwhm=6
smoothed = data.smooth(fwhm=fwhm)
```

Let's take a look and see how this changes the image.

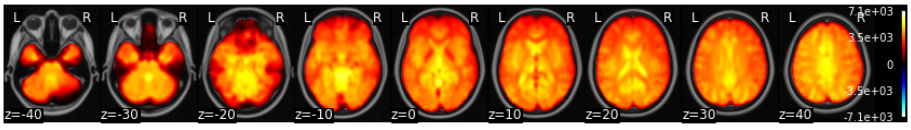
```
data.mean().plot()
```

```
threshold is ignored for simple axial plots
```




```
smoothed.mean().plot()
```

```
threshold is ignored for simple axial plots
```



Estimate GLM for all voxels

Now we are ready to estimate the regression model for all voxels.

We will assign the design_matrix object to the `.X` attribute of our `Brain_Data` instance.

Then we simply need to run the `.regress()` method.

```
smoothed.X = dm_conv_filt_poly_cov
stats = smoothed.regress()

print(stats.keys())
```

```
dict_keys(['beta', 't', 'p', 'sigma', 'residual'])
```

Ok, it's done! Let's take a look at the results.

The stats variable is a dictionary with the main results from the regression: a brain image with all of the betas for each voxel, a correspondign image of t-values, p-values, standard error of the estimate, and residuals.

Remember we have run the same regression model separately on each voxel of the brain.

Let's take a look at one of the regressors. The names of each of them are in the column names of the design matrix, which is in the `data.X` field. We can print them to see the names. Let's plot the first one, which is a horizontal checkerboard.

```
print(smoothed.X.columns)
```

```
Index(['video_computation_c0', 'horizontal_checkerboard_c0',
       'audio_right_hand_c0', 'audio_sentence_c0', 'video_right_hand_c0',
       'audio_left_hand_c0', 'video_left_hand_c0', 'vertical_checkerboard_c0',
       'audio_computation_c0', 'video_sentence_c0', 'cosine_1', 'cosine_2',
       'cosine_3', 'cosine_4', 'poly_0', 'poly_1', 'poly_2', 'poly_3',
       'trans_x', 'trans_y', 'trans_z', 'rot_x', 'rot_y', 'rot_z', 'trans_x',
       'trans_y', 'trans_z', 'rot_x', 'rot_y', 'rot_z', 'trans_x', 'trans_y',
       'trans_z', 'rot_x', 'rot_y', 'rot_z', 'trans_x', 'trans_y', 'trans_z',
       'rot_x', 'rot_y', 'rot_z', 'diff_spike1', 'diff_spike2', 'diff_spike3'],
      dtype='object')
```

Brain_Data instances have their own plotting methods. We will be using `.iplot()` here, which can allow us to interactively look at all of the values.

If you would like to see the top values, we can quickly apply a threshold. Try using `95`% threshold, and be sure to click the `percentile_threshold` option.

```
stats['beta'][0].iplot()
```

Save Image

We will frequently want to save different brain images we are working with to a nifti file. This is useful for saving intermediate work, or sharing our results with others. This is easy with the `.write()` method. Be sure to specify a path and file name for the file.

Note: You can only write to folders where you have permission. Try changing the path to your own directory.

```
smoothed.write(f'{sub}_betas_denoised_smoothed{fwhm}_preprocessed_fMRI_bold.nii.gz')
```

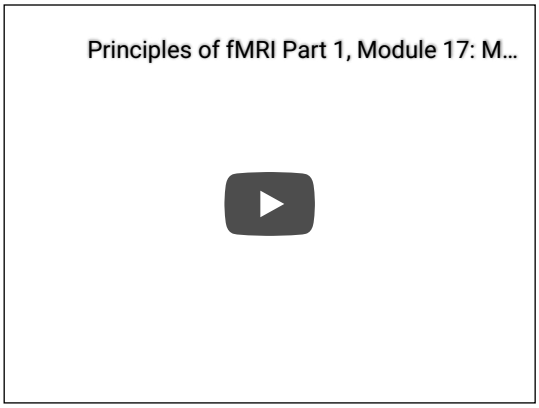
Contrasts

Now that we have estimated our model, we will likely want to create contrasts to examine brain activation to different conditions.

This procedure is identical to those introduced in our GLM tutorial.

Let's watch another video by Tor Wager to better understand contrasts at the first-level model stage.

```
YouTubeVideo('7MibM1ATai4')
```



Now, let's try making a simple contrast where we average only the regressors pertaining to motor. This is essentially summing all of the motor regressors. To take the mean we need to divide by the number of regressors.

```
print(smoothed.X.columns)

c1 = np.zeros(len(stats['beta']))
c1[[2,4,5,6]] = 1/4
print(c1)

motor = stats['beta'] * c1

motor.iplot()
```

```
Index(['video_computation_c0', 'horizontal_checkerboard_c0',
       'audio_right_hand_c0', 'audio_sentence_c0', 'video_right_hand_c0',
       'audio_left_hand_c0', 'video_left_hand_c0', 'vertical_checkerboard_c0',
       'audio_computation_c0', 'video_sentence_c0', 'cosine_1', 'cosine_2',
       'cosine_3', 'cosine_4', 'poly_0', 'poly_1', 'poly_2', 'poly_3',
       'trans_x', 'trans_y', 'trans_z', 'rot_x', 'rot_y', 'rot_z', 'trans_x',
       'trans_y', 'trans_z', 'rot_x', 'rot_y', 'rot_z', 'trans_x', 'trans_y',
       'trans_z', 'rot_x', 'rot_y', 'rot_z', 'trans_x', 'trans_y', 'trans_z',
       'rot_x', 'rot_y', 'rot_z', 'diff_spike1', 'diff_spike2', 'diff_spike3'],
      dtype='object')
[0.   0.   0.25 0.   0.25 0.25 0.25 0.   0.   0.   0.   0.   0.   0.
 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
 0.   0.   0.  ]
```

Ok, now we can clearly see regions specifically involved in motor processing.

Now let's see which regions are more active when making motor movements with our right hand compared to our left hand.

```
c_rvl = np.zeros(len(stats['beta']))
c_rvl[[2,4,5,6]] = [.5, .5, -.5, -.5]

motor_rvl = stats['beta'] * c_rvl

motor_rvl.iplot()
```

What do you see?

Exercises

For homework, let's get a better handle on how to play with our data and test different hypotheses.

1. Which regions are more involved with visual compared to auditory sensory processing?

- Create a contrast to test this hypothesis
- plot the results
- write the file to your output folder.

2. Which regions are more involved in processing numbers compared to words?

- Create a contrast to test this hypothesis
- plot the results
- write the file to your output folder.

3. Which regions are more involved with motor compared to cognitive processes (e.g., language and math)?

- Create a contrast to test this hypothesis
- plot the results
- write the file to your output folder.

4. How are your results impacted by different smoothing kernels?

- Pick two different sized smoothing kernels and create two new brain images with each smoothing kernel
- Pick any contrast of interest to you and evaluate the impact of smoothing on the contrast.
- plot the results
- write the file to your output folder.

Group Analysis

Written by Luke Chang

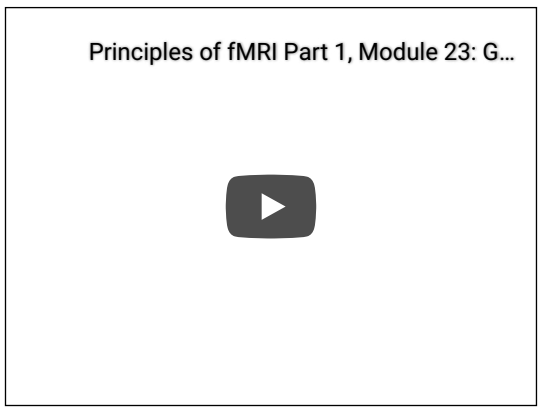
In fMRI analysis, we are primarily interested in making inferences about how the brain processes information that is fundamentally similar across all brains even for people that did not directly participate in our study. This requires making inferences about the magnitude of the population level brain response based on measurements from a few randomly sampled participants who were scanned during our experiment.

In this tutorial, we will cover how we go from modeling brain responses in each voxel for a single participant to making inferences about the group. We will cover the following topics:

- Mixed Effects Models
- How to use the summary statistic approach to make inferences at second level
- How to perform many types of inferences at second level with different types of design matrics

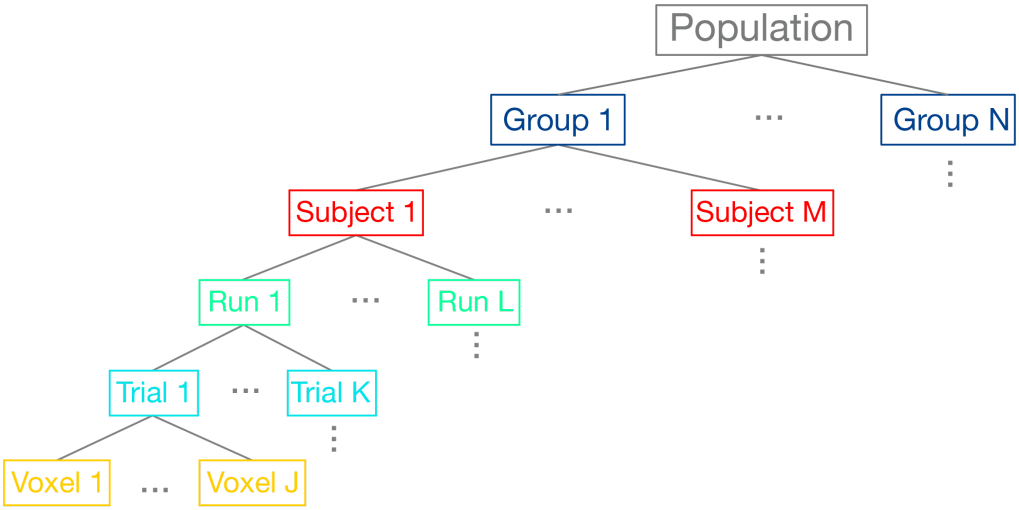
Let's start by watching an overview of group statistics by Tor Wager.

```
from IPython.display import YouTubeVideo

YouTubeVideo('__cOYPifDWk')
```



Hierarchical Data Structure

We can think of the data as being organized into a hierarchical structure. For each brain, we are measuring BOLD activity in hundreds of thousands of cubic voxels sampled at about 0.5Hz (i.e., TR=2s). Our experimental task will have many different trials for each condition (seconds), and these trials may be spread across multiple scanning runs (minutes), or entire scanning sessions (hours). We are ultimately interested in modeling all of these different scales of data to make an inference about the function of a particular region of the brain across the group of participants we sampled, which we would hope will generalize to the broader population.

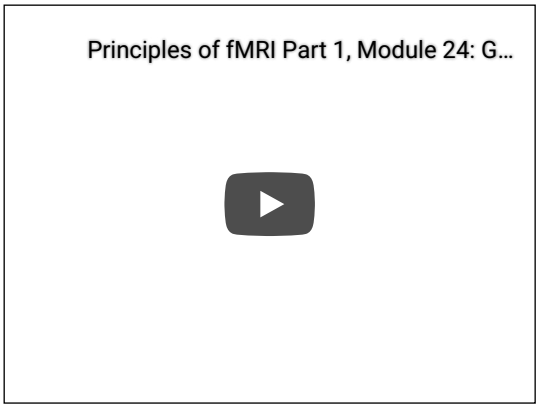


In the past few notebooks, we have explored how to preprocess the data to reduce noise and enhance our signal and also how we can estimate responses in each voxel to specific conditions within a single participant based on convolving our experimental design with a canonical hemodynamic response function (HRF). Here we will discuss how we combine these brain responses estimated at the first-level in a second-level model to make inferences about the group.

Modeling Mixed Effects

Let's dive deeper into how we can model both random and fixed effects using multi-level models by watching another video by Tor Wager.

```
YouTubeVideo('-abMLQSjMSI')
```



Most of the statistics we have discussed to this point have assumed that the data we are trying to model are drawn from an identical distribution and that they are independent of each other. For example, each group of participants that complete each version of our experiment are assumed to be random sample of the larger population. However, if there was some type of systematic bias in our sampling strategy, our group level statistics would not necessarily reflect a random draw from the population-level Gaussian distribution. However, as should already be clear from the graphical depiction of the hierarchical structure of our data above, our data are not always independent. For example, we briefly discussed this in the GLM notebook, but voxel responses within the same participant are not necessarily independent as there appears to be a small amount of autocorrelation in the BOLD response. This requires whitening the data to meet the independence assumption. What is clear from the hierarchy is that all of the data measured from one participant are likely to be more similar to each other than another participant. In fact, it is almost always the case that the variance *within* a subject σ^2_{within} is almost always smaller than the variance *across* participants $\sigma^2_{between}$. If we combined all of the data from all participants and treated them as if they were independent, we would likely have an inflated view of the group effect (this was historically referred to as a "fixed effects group analysis").

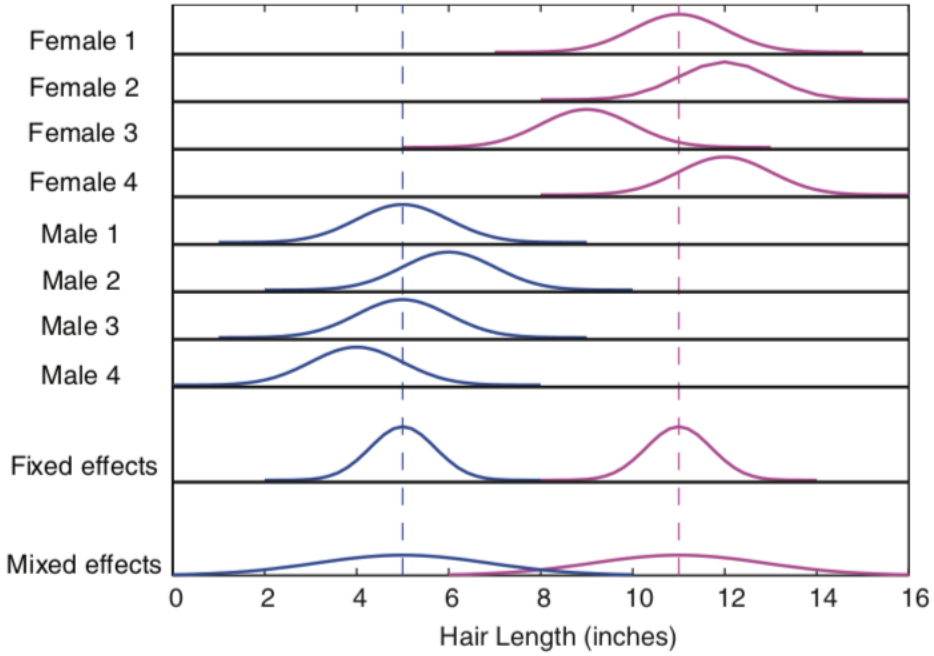
This problem has been elegantly solved in statistics in a class of models called *mixed effects models*. Mixed effects models are an extension of regression that allows data to be structured into groups and coefficients to vary by groups. They are referred to differently in different scientific domains, for example they may be referred to as

multilevel, hierarchical, or panel models. The reason that this framework has been found to be useful in many different fields, is that it is particularly well suited for modeling clustered data, such as students in a classroom and also longitudinal or repeated data, such as within-subject designs.

The term “mixed” comes from the fact that these models are composed of both *fixed* and *random* effects. Fixed effects refer to parameters describing the amount of variance that a feature explains of an outcome variable. Fixed factors are often explicitly manipulated in an experiment and can be categorical (e.g., gender) or continuous (e.g., age). We assume that the magnitude of these effects are *fixed* in the population, but that the observed signal strength will vary across sessions and subjects. This variation can be decomposed into different sources of variance, such as: - Measurement or Irreducible Error - Response magnitude that varies randomly across subjects. - Response magnitude that varies randomly across different elicitations (e.g., trials or sessions).

Modeling these different sources of variance allows us to have a better idea of how generalizable our estimates might be to another participant or trial.

As an example, imagine if we were interested if there were any gender differences between the length of how males and females cut their hair. We might sample a given individual several times over the course of a couple of years to get an accurate measurement of how long they keep their hair. These samples are akin to trials and will give us a way to represent the overall tendency of the length an individual keeps their hair in the form of a distribution. Narrow distributions mean that there is little variability in the length of the hair at each measurement, while wider distributions indicate more variation in the hair length across time. Of course, we are most interested not in the length of how an individual cuts their hair, but rather how many individuals from the same group cut their hair. This requires measuring multiple participants, who will all vary randomly around some population level hair length parameter. We are interested in modeling the true *fixed effect* of what the population parameter is for hair length, and specifically, whether this differs across gender. The variation in measurements within an individual and across individuals will reflect some degree of randomness that we need to account for in order to estimate a parameter that will generalize beyond the participants we measured their hair, but to new participants.

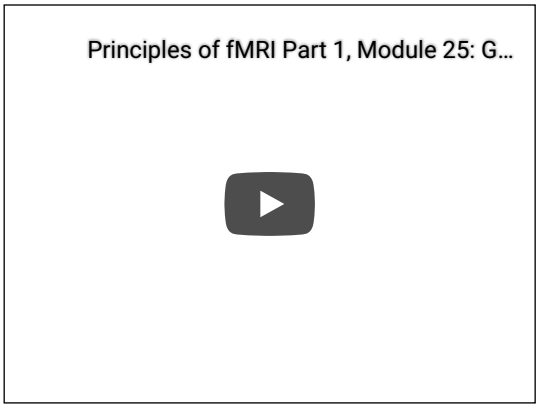


from Poldrack, Mumford, & Nichols (2011)

In statistics, it is useful to distinguish between the *model* used to describe the data, the *method* of parameter estimation, and the *algorithm* used to obtain them.

Let’s now watch a video by Martin Lindquist to learn more about the way these models are estimated.

```
YouTubeVideo('-yaHTygR9b8')
```



First Level - Single Subject Model

In fMRI data analysis, we often break analyses into multiple stages. First, we are interested in estimating the parameter (or distribution) of signal in a given region resulting from our experimental manipulation, while simultaneously attempting to control for as much noise and artifacts as possible. This will give us a a single number for each participant of the average length they keep their hair.

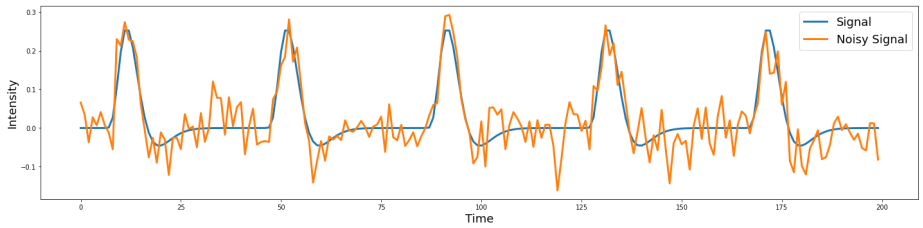
At the first level model, for each participant we can define our model as:

$Y_i = X_i\beta + \epsilon_i$, where i is an observation for a single participant and $\epsilon_i \sim \mathcal{N}(0, \sigma_i^2)$

Because participants are independent, it is possible to estimate each participant separately.

To provide a concrete illustration of the different sources of variability in a signal, let's make a quick simulation a hypothetical voxel timeseries.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.stats import regress, zscore
from nltools.data import Brain_Data, Design_Matrix
from nltools.stats import regress
from nltools.external import glover_hrf
from scipy.stats import ttest_1samp

def plot_timeseries(data, linewidth=3, labels=None, axes=True):
    f,a = plt.subplots(figsize=(20,5))
    a.plot(data, linewidth=linewidth)
    a.set_ylabel('Intensity', fontsize=18)
    a.set_xlabel('Time', fontsize=18)
    plt.tight_layout()
    if labels is not None:
        plt.legend(labels, fontsize=18)
    if not axes:
        a.axes.get_xaxis().set_visible(False)
        a.axes.get_yaxis().set_visible(False)

def simulate_timeseries(n_tr=200, n_trial=5, amplitude=1, tr=1, sigma=0.05):
    y = np.zeros(n_tr)
    y[np.arange(20, n_tr, int(n_tr/n_trial))] = amplitude

    hrf = glover_hrf(tr, oversampling=1)
    y = np.convolve(y, hrf, mode='same')
    epsilon = sigma*np.random.randn(n_tr)
    y = y + epsilon
    return y

sim1 = simulate_timeseries(sigma=0)
sim2 = simulate_timeseries(sigma=0.05)
plot_timeseries(np.vstack([sim1,sim2]).T, labels=['Signal', 'Noisy Signal'])
```



Notice that the noise appears to be independent over each TR.

Second level summary of between group variance

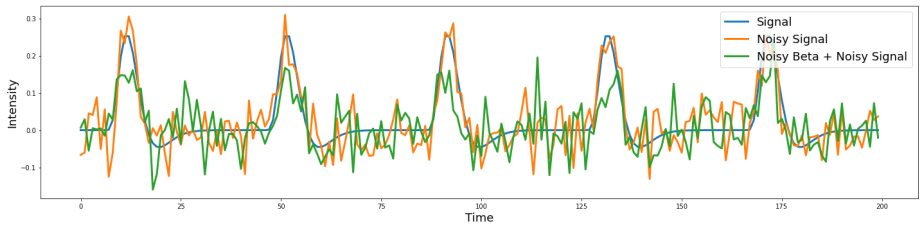
In the second level model, we are interested in relating the subject specific parameters contained in β to the population parameters β_g. We assume that the first level parameters are randomly sampled from a population of possible regression parameters.

$$\beta = X_g \beta_g + \eta$$

$$\eta \sim \mathcal{N}(0,\ \sigma_g^2)$$

Now let's add noise onto the beta parameter to see what happens.

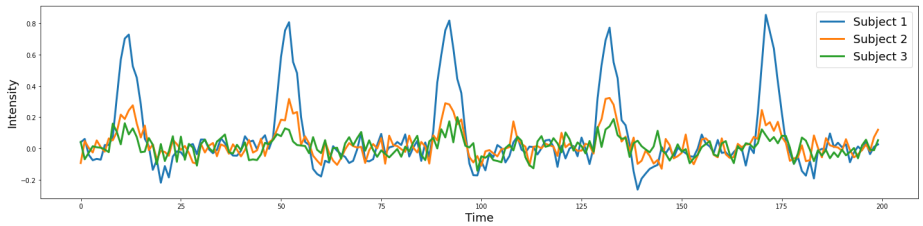
```
beta = np.abs(np.random.randn())*3
sim1 = simulate_timeseries(sigma=0)
sim2 = simulate_timeseries(sigma=0.05)
sim3 = simulate_timeseries(amplitude=beta, sigma=0.05)
plot_timeseries(np.vstack([sim1,sim2,sim3]).T, labels=['Signal', 'Noisy Signal', 'Noisy
Beta + Noisy Signal'])
```



Try running the above code several times. Can you see how the beta parameter impacts the amplitude of each trial, while the noise appears to be random and uncorrelated with the signal?

Let's try simulating three subjects with a beta drawn from a normal distribution.

```
sim1 = simulate_timeseries(amplitude=np.abs(np.random.randn())*2, sigma=0.05)
sim2 = simulate_timeseries(amplitude=np.abs(np.random.randn())*2, sigma=0.05)
sim3 = simulate_timeseries(amplitude=np.abs(np.random.randn())*2, sigma=0.05)
plot_timeseries(np.vstack([sim1, sim2, sim3]).T, labels=['Subject 1', 'Subject 2',
'Subject 3'])
```



To make an inference if there is a reliable difference within or across groups, we need to model the distribution of the parameters resulting from the first level model using a second level model. For example, if we were solely interested in estimating the average length men keep their hair, we would need to measure hair lengths from lots of different men and the average would be our best guess for any new male sampled from the same population. In our example, we are explicitly interested in the pairwise difference between males and females in hair length. Does the mean hair length for one sex significantly different from the hair length of the other group that is larger than the variations in hair length we observe within each group?

Mixed Effects Model

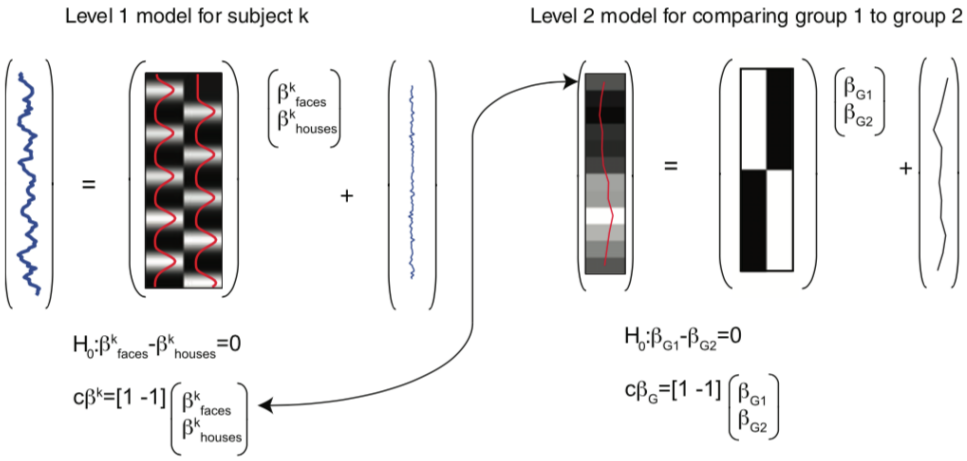
In neuroimaging data analysis, there are two main approaches to implementing these different models. Some software packages attempt to use a computationally efficient approximation and use what is called a two stage summary statistic approach. First level models are estimated separately for every participant and then the betas from each participant's model is combined in a second level model. This is the strategy implemented in SPM and is computationally efficient. However, another approach simultaneously estimates the first and second level models at the same time and often use algorithms that iterate back and forth from the single to the group. The main advantage of this approach over the two-stage approach is that the uncertainty in the parameter estimates at the first-level can be appropriately weighted at the group level. For example, if we had a bad participant with very noisy data, we might not want to weight their estimate when we aggregate everyone's data across the group. The disadvantage of this approach is that the estimation procedure is considerably more computationally expensive. This is the approach implemented in FSL, BrainVoyager, and AFNI. In practice, the advantage of the true random effects simultaneous parameter estimation only probably benefits getting more reliable estimates when the sample size is small. In the limit, both methods should converge to the same answer. For a more in depth comparison see this blog post by Eshin Jolly.

A full mixed effects model can be written as,

$$Y_i = X_i(X_g\beta_g + \eta) + \epsilon_i$$

```
or
```

$$Y \sim (XX_g\beta_g, X\sigma_g^2X^T + \sigma^2)$$

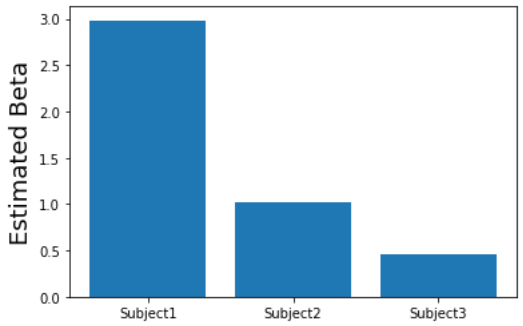


from Poldrack, Mumford, & Nichols (2011)

Let's now try to recover the beta estimates from our 3 simulated subjects.

```
# Create a design matrix with an intercept and predicted response
task = simulate_timeseries(amplitude=1, sigma=0)
X = np.vstack([np.ones(len(task)), task]).T

# Loop over each of the simulated participants and estimate the amplitude of the
response.
betas = []
for sub in [sim1, sim2, sim3]:
    beta,_,_,_,_ = regress(X, sub)
    betas.append(beta[1])

# Plot estimated amplitudes for each participant
plt.bar(['Subject1', 'Subject2', 'Subject3'], betas)
plt.ylabel('Estimated Beta', fontsize=18)
```

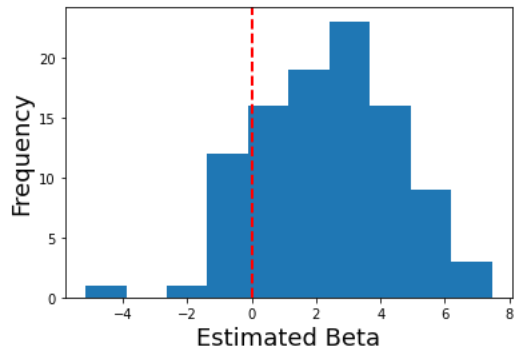
```
Text(0, 0.5, 'Estimated Beta')
```



What if we simulated lots of participants? What would the distribution of betas look like?

```
# Create a design matrix with an intercept and predicted response
task = simulate_timeseries(amplitude=1, sigma=0)
X = np.vstack([np.ones(len(task)), task]).T

# Loop over each of the simulated participants and estimate the amplitude of the
response.
betas = []
for sub in range(100):
    sim = simulate_timeseries(amplitude=2+np.random.randn()*2, sigma=0.05)
    beta,_,_,_,_ = regress(X, sim)
    betas.append(beta[1])

# Plot distribution of estimated amplitudes for each participant
plt.hist(betas)
plt.ylabel('Frequency', fontsize=18)
plt.xlabel('Estimated Beta', fontsize=18)
plt.axvline(x=0, color='r', linestyle='dashed', linewidth=2)
```

```
<matplotlib.lines.Line2D at 0x7f990c25ced0>
```



Now in a second level analysis, we are interested in whether there is a reliable effect across all participants in our sample. In other words, is there a response to our experiment for a specific voxel that is reliably present across our sample of participants?

We can test this hypothesis in our simulation by running a one-sample ttest across the estimated first-level betas at the second level. This allows us to test whether the sample has signal that is reliably different from zero (i.e., the null hypothesis).

```
ttest_1samp(betas, 0)
```

```
Ttest_1sampResult(statistic=10.854776909716737, pvalue=1.50775361636617e-18)
```

What did we find?

Running a Group Analysis

Okay, now let's try and run our own group level analysis with real imaging data using the Pinel Localizer data. I have run a first level model for the first 10 participants using the procedure we used in the single-subject analysis notebook.

Here is the code I used to complete this for all participants. I wrote all of the betas and also a separate file for each individual regressor of interest.

```
import os
from glob import glob
import pandas as pd
import numpy as np
import nibabel as nib
from nltools.stats import zscore, regress, find_spikes
from nltools.data import Brain_Data, Design_Matrix
from bids import BIDSLayout, BIDSValidator
from nltools.file_reader import onsets_to_dm
from nltools.data import Brain_Data, Design_Matrix
from nilearn.plotting import view_img, glass_brain, plot_stat_map

data_dir = '../data/localizer'
layout = BIDSLayout(data_dir, derivatives=True)

tr = layout.get_tr()
fwhm = 6
spike_cutoff = 3

def load_bids_events(layout, subject):
    '''Create a design_matrix instance from BIDS event file'''

    tr = layout.get_tr()
    n_tr = nib.load(layout.get(subject=subject, scope='raw', suffix='bold')[0].path).shape[-1]

    onsets = pd.read_csv(layout.get(subject=subject, suffix='events')[0].path, sep='\t')
    onsets.columns = ['Onset', 'Duration', 'Stim']
    return onsets_to_dm(onsets, sampling_freq=1/tr, run_length=n_tr)

def make_motion_covariates(mc):
    z_mc = zscore(mc)
    all_mc = pd.concat([z_mc, z_mc**2, z_mc.diff(), z_mc.diff()**2], axis=1)
    all_mc.fillna(value=0, inplace=True)
    return Design_Matrix(all_mc, sampling_freq=1/tr)

for sub in layout.get_subjects(scope='derivatives'):
    data = Brain_Data([x for x in layout.get(subject=sub, scope='derivatives', suffix='bold',
extension='nii.gz', return_type='file') if 'denoised' not in x][0])
    smoothed = data.smooth(fwhm=fwhm)

    dm = load_bids_events(layout, sub)
    covariates = pd.read_csv(layout.get(subject=sub, scope='derivatives', extension='.tsv')
[0].path, sep='\t')
    mc_cov = make_motion_covariates(covariates[['trans_x','trans_y','trans_z','rot_x',
'rot_y', 'rot_z']])
    spikes = data.find_spikes(global_spike_cutoff=spike_cutoff,
diff_spike_cutoff=spike_cutoff)
    dm_cov = dm.convolve().add_dct_basis(duration=128).add_poly(order=1, include_lower=True)
    dm_cov = dm_cov.append(mc_cov, axis=1).append(Design_Matrix(spikes.iloc[:, 1:],
sampling_freq=1/tr), axis=1)
    smoothed.X = dm_cov
    stats = smoothed.regress()
    file_name = layout.get(subject=sub, scope='derivatives', suffix='bold',
extension='nii.gz', return_type='file')[0]
    stats['beta'].write(os.path.join(os.path.dirname(file_name), f"sub-
{sub}_betas_denoised_{file_name.split('_')[1]}_{file_name.split('_')
[2]}_smoothed{fwhm}_{file_name.split('_')[-1]}"))

    for i, name in enumerate([x[:-3] for x in dm_cov.columns[:10]]):
        stats['beta'][i].write(os.path.join(os.path.dirname(file_name), f"sub-
{sub}_{name}_denoised_{file_name.split('_')[2]}_smoothed{fwhm}_{file_name.split('_')[-1]}"))
```

Now, we are ready to run our first group analyses!

Let's load our design matrix to remind ourselves of the various conditions

One Sample t-test

For our first group analysis, let's try to examine which regions of the brain are consistently activated across participants. We will just load the first regressor in the design matrix - *horizontal_checkerboard*.

We will use the `glob` function to search for all files that contain the name *horizontal_checkerboard* in each subject's folder. We will then sort the list and load all of the files using the `Brain_Data` class. This will take a little bit to load all of the data into ram.

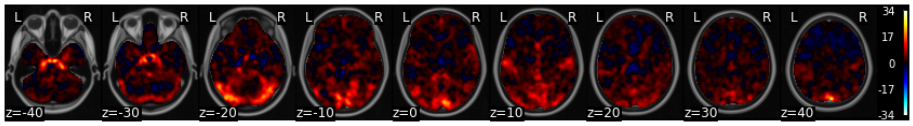
```
import glob

con1_name = 'horizontal_checkerboard'
con1_file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep','*', 'func',
f'sub*_{con1_name}*nii.gz'))
con1_file_list.sort()
con1_dat = Brain_Data(con1_file_list)
```

Now that we have the data loaded, we can run quick operations such as, what is the mean activation in each voxel across participants? Or, what is the standard deviation of the voxel activity across participants?

Notice how we can chain different commands like `.mean()` and `.plot()`. This makes it easy to quickly manipulate the data similar to how we use tools like pandas.

```
con1_dat.mean().plot()
```

```
threshold is ignored for simple axial plots
```



We can use the `ttest()` method to run a quick t-test across each voxel in the brain.

```
con1_stats = con1_dat.ttest()

print(con1_stats.keys())
```

```
dict_keys(['t', 'p'])
```

This return a dictionary of a map of the t-values and a separate one containing the p-value for each voxel.

For now, let's look at the results of the t-ttest and threshold them to something like t>4.

```
con1_stats['t'].iplot()
```

As you can see we see very clear activation in various parts of visual cortex, which we expected from the visual stimulation.

However, if wanted to test the hypothesis that there are specific areas of early visual cortex (e.g., V1) that process edge orientations, we could run a specific contrast comparing vertical orientations with horizontal orientations.

Now we need to load the vertical data and create a contrast between horizontal and vertical checkerboards.

Here a contrast is simply [1, -1] and can be achieved by simply subtracting the two images (assuming the subject images are sorted in the same order).

```
con2_name = 'vertical_checkerboard'
con2_file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep','*', 'func',
f'sub*_{con2_name}*nii.gz'))
con2_file_list.sort()
con2_dat = Brain_Data(con2_file_list)

con1_v_con2 = con1_dat-con2_dat
```

Again, we will now run a one-sample ttest on the contrast to find regions that are consistently different in viewing horizontal vs vertical checkerboards across participants at the group level.

```
con1_v_con2_stats = con1_v_con2.ttest()
con1_v_con2_stats['t'].iplot()
```

Group statistics using design matrices

For these analyses we ran a one-sample t-test to examine the average activation to horizontal checkerboards and the difference between viewing horizontal and vertical checkerboards. This is equivalent to a vector of ones at the second level. The latter analysis is technically a paired-samples t-test.

Do these tests sound familiar?

It turns out that most parametric statistical tests are just special cases of the general linear model. Here are what the design matrices would look like for various types of statistical tests.

Test Description	Order of data	Xβ	Hypothesis Test
One-sample t-test. 6 observations	G_1 G_2 G_3 G_4 G_5 G_6	$\begin{pmatrix}1\\1\\1\\1\\1\\1\end{pmatrix}\begin{bmatrix}\beta_1\end{bmatrix}$	H_0: Overall mean=0 $H_0: \beta_1 = 0$ $H_0: c\beta = 0$ $c = [1]$
Two-sample t-test. 5 subjects in group 1 (G1) and 5 subjects in group 2 (G2)	$G1_1$ $G1_2$ $G1_3$ $G1_4$ $G1_5$ $G2_1$ $G2_2$ $G2_3$ $G2_4$ $G2_5$	$\begin{pmatrix}1&0\\1&0\\1&0\\1&0\\1&0\\0&1\\0&1\\0&1\\0&1\\0&1\end{pmatrix}\begin{pmatrix}\beta_{G1}\\\beta_{G2}\end{pmatrix}$	H_0: mean of G1 different from G2 $H_0: \beta_{G1} - \beta_{G2} = 0$ $H_0: c\beta = 0$ $c = [1\ \ -1]$
Paired t-test. 5 paired measures of A and B.	A_{S1} B_{S1} A_{S2} B_{S2} A_{S3} B_{S3} A_{S4} B_{S4} A_{S5} B_{S5}	$\begin{pmatrix}1&1&0&0&0&0\\-1&1&0&0&0&0\\1&0&1&0&0&0\\-1&0&1&0&0&0\\1&0&0&1&0&0\\-1&0&0&1&0&0\\1&0&0&0&1&0\\-1&0&0&0&1&0\\1&0&0&0&0&1\\-1&0&0&0&0&1\end{pmatrix}\begin{pmatrix}\beta_{diff}\\\beta_{S1}\\\beta_{S2}\\\beta_{S3}\\\beta_{S4}\\\beta_{S5}\end{pmatrix}$	H_0: A is different from B $H_0: \beta_{diff} = 0$ $H_0: c\beta = 0$ $c = [1\ 0\ 0\ 0\ 0\ 0]$
Two way ANOVA. Factor A has two levels and factor B has 3 levels. There are 2 observations for each A/B combination.	$A1B1_1$ $A1B1_2$ $A1B2_1$ $A1B2_2$ $A1B3_1$ $A1B3_2$ $A2B1_1$ $A2B1_2$ $A2B2_1$ $A2B2_2$ $A2B3_1$ $A2B3_2$	$\begin{pmatrix}1&1&1&0&1&0\\1&1&1&0&1&0\\1&1&0&1&0&1\\1&1&0&1&0&1\\1&1&-1&-1&-1&-1\\1&1&-1&-1&-1&-1\\1&-1&1&0&-1&0\\1&-1&1&0&-1&0\\1&-1&0&1&0&-1\\1&-1&0&1&0&-1\\1&-1&-1&-1&1&1\\1&-1&-1&-1&1&1\end{pmatrix}\begin{pmatrix}\beta_{mean}\\\beta_{A1}\\\beta_{B1}\\\beta_{B2}\\\beta_{A1B1}\\\beta_{A1B2}\end{pmatrix}$	F-tests for all contrasts H_0: Overall mean=0 $H_0: \beta_{mean} = 0$ $H_0: c\beta = 0$ $c = [1\ 0\ 0\ 0\ 0\ 0]$ H_0: Main A effect=0 $H_0: \beta_{A1} = 0$ $H_0: c\beta = 0$ $c = [0\ 1\ 0\ 0\ 0\ 0]$ H_0: Main B effect = 0 $H_0: \beta_{B1} = \beta_{B2} = 0$ $H_0: c\beta = 0$ $c = \begin{bmatrix}0&0&1&0&0&0\\0&0&0&1&0&0\end{bmatrix}$ H_0: A/B interaction effect=0 $H_0: \beta_{A1B1} = \beta_{A1B2} = 0$ $H_0: c\beta = 0$ $c = \begin{bmatrix}0&0&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}$

from Poldrack, Mumford, & Nichols 2011

In this section, we will explore how we can formulate different types of statistical tests using a regression through simulations.

One Sample t-test

Just to review, our one sample t-test can also be formulated as a regression, where the beta values for each subject in a voxel are predicted by a vector of ones. This *intercept* only model, computes the mean of y. If the mean of y (i.e., the intercept) is consistently shifted away from zero, then we can reject the null hypothesis that the mean of the betas is zero.

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} \beta_0 \end{bmatrix}$$

We can simulate this by generating data from a Gaussian distribution. We will generate two groups, where y reflects equal draws from each of these distributions $group_1 = \mathcal{N}(10, 2)$ and $group_2 = \mathcal{N}(5, 2)$. We then regress a vector of ones on y.

We report the estimated value of beta and compare it to various summaries of the simulated data. This allows us to see exactly what each parameter in the regression is calculating.

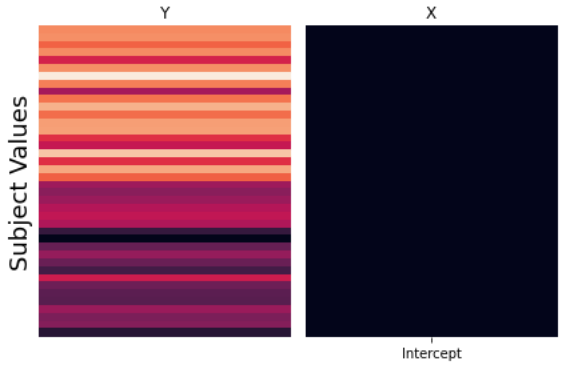
First, let's define a function `run_regression_simulation` to help us generate plots and calculate various ways to summarize the simulation.

```
def run_regression_simulation(x, y, paired=False):
    '''This Function runs a regression and outputs results'''
    # Estimate Regression
    if not paired:
        b, t, p, df, res = regress(x, y)
        print(f"betas: {b}")
        if x.shape[1] > 1:
            print(f"beta1 + beta2: {b[0] + b[1]}")
            print(f"beta1 - beta2: {b[0] - b[1]}")
            print(f"mean(group1): {np.mean(group1)}")
            print(f"mean(group2): {np.mean(group2)}")
            print(f"mean(group1) - mean(group2): {np.mean(group1)-np.mean(group2)}")
        print(f"mean(y): {np.mean(y)}")
    else:
        beta, t, p, df, res = regress(x, y)
        a = y[x.iloc[:,0]==1]
        b = y[x.iloc[:,0]==-1]
        out = []
        for sub in range(1, X.shape[1]):
            sub_dat = y[X.iloc[:, sub]==1]
            out.append(sub_dat-np.mean(sub_dat))
        avg_sub_mean_diff = np.mean([x[0] for x in out])
        print(f"betas: {b}")
        print(f"contrast beta: {beta[0]}")
        print(f"mean(subject betas): {np.mean(beta[1:])}")
        print(f"mean(y): {np.mean(y)}")
        print(f"mean(a): {a.mean()}")
        print(f"mean(b): {b.mean()}")
        print(f"mean(a-b): {np.mean(a - b)}")
        print(f"sum(a_i-mean(y_i))/n: {avg_sub_mean_diff}")

    # Create Plot
    f,a = plt.subplots(ncols=2, sharey=True)
    sns.heatmap(pd.DataFrame(y), ax=a[0], cbar=False, yticklabels=False,
xticklabels=False)
    sns.heatmap(x, ax=a[1], cbar=False, yticklabels=False)
    a[0].set_ylabel('Subject Values', fontsize=18)
    a[0].set_title('Y')
    a[1].set_title('X')
    plt.tight_layout()
```

okay, now let's run the simulation for the one sample t-test.

```
group1_params = {'n':20, 'mean':10, 'sd':2}
group2_params = {'n':20, 'mean':5, 'sd':2}
group1 = group1_params['mean'] + np.random.randn(group1_params['n']) *
group1_params['sd']
group2 = group2_params['mean'] + np.random.randn(group2_params['n']) *
group2_params['sd']

y = np.hstack([group1, group2])
x = pd.DataFrame({'Intercept':np.ones(len(y))})

run_regression_simulation(x, y)
```

```
betas: 7.767465428733612
mean(y): 7.767465428733613
```



The results of this simulation clearly demonstrate that the intercept of the regression is modeling the mean of y.

Independent-Samples T-Test - Dummy Codes

Next, let's explore how we can compute an independent-sample t-test using a regression. There are several different ways to compute this. Each of them provides a different way to test for differences between the means of the two samples.

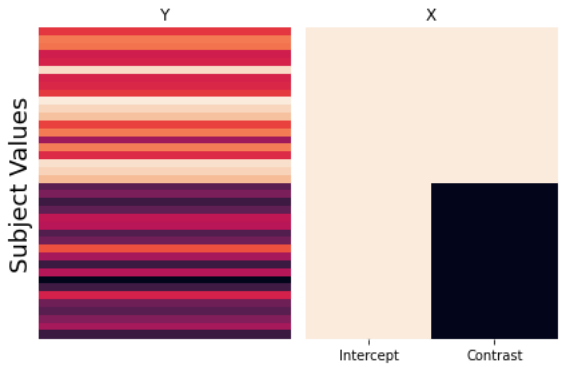
First, we will explore how dummy codes can be used to test for group differences. We will create a design matrix with an intercept and also a column with a binary regressor indicating group membership. The target group will be ones, and the reference group will be zeros.

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

Let's run another simulation examining what the regression coefficients reflect using this dummy code approach.

```
group1_params = {'n':20, 'mean':10, 'sd':2}
group2_params = {'n':20, 'mean':5, 'sd':2}
group1 = group1_params['mean'] + np.random.randn(group1_params['n']) *
group1_params['sd']
group2 = group2_params['mean'] + np.random.randn(group2_params['n']) *
group2_params['sd']

y = np.hstack([group1, group2])
x = pd.DataFrame({'Intercept':np.ones(len(y)),
'Contrast':np.hstack([np.ones(group1_params['n']), np.zeros(group2_params['n'])])})

run_regression_simulation(x, y)
```

```
betas: [5.59590636 4.86905406]
beta1 + beta2: 10.46496042000535
beta1 - beta2: 0.7268523061306578
mean(group1): 10.464960420005351
mean(group2): 5.595906363068004
mean(group1) - mean(group2): 4.869054056937347
mean(y): 8.030433391536677
```



Can you figure out what the beta estimates are calculating?

The intercept β_0 is now the mean of the reference group, and the estimate of the dummy code regressor β_1 indicates the difference of the mean of the target group from the reference group.

Thus, the mean of the reference group is β_0 or the intercept, and the mean of the target group is $\beta_1 + \beta_2$.

Independent-Samples T-Test - Contrasts

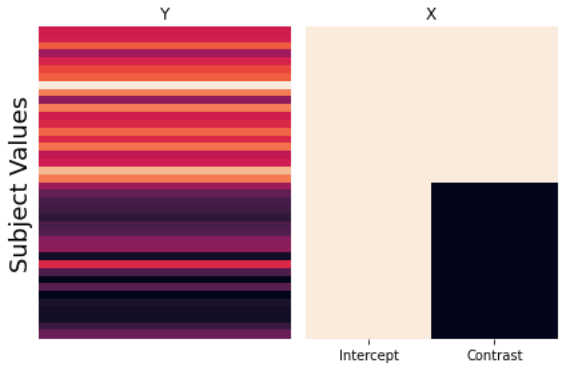
Another way to compare two different groups is by creating a model with an intercept and contrast between the two groups.

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

Let's now run another simulation to see how these beta estimates differ from the dummy code model.

```
group1_params = {'n':20, 'mean':10, 'sd':2}
group2_params = {'n':20, 'mean':5, 'sd':2}
group1 = group1_params['mean'] + np.random.randn(group1_params['n']) *
group1_params['sd']
group2 = group2_params['mean'] + np.random.randn(group2_params['n']) *
group2_params['sd']

y = np.hstack([group1, group2])
x = pd.DataFrame({'Intercept':np.ones(len(y)),
'Contrast':np.hstack([np.ones(group1_params['n']), -1*np.ones(group2_params['n'])])})

run_regression_simulation(x, y)
```

```
betas: [7.60615263 3.16144295]
beta1 + beta2: 10.767595577579165
beta1 - beta2: 4.444709673800184
mean(group1): 10.767595577579163
mean(group2): 4.4447096738001814
mean(group1) - mean(group2): 6.322885903778982
mean(y): 7.606152625689674
```



So, just as before, the intercept reflects the mean of y. Now can you figure out what β_1 is calculating?

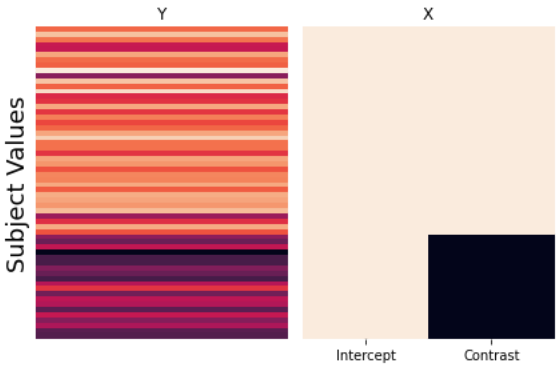
It is the average distance of each group to the mean. The mean of group 1 is $\beta_0 + \beta_1$ and the mean of group 2 is $\beta_0 - \beta_1$.

Remember that in our earlier discussion of contrast codes, we noted the importance of balanced codes across regressors. What if the group sizes are unbalanced? Will this effect our results?

To test this, we will double the sample size of group1 and rerun the simulation.

```
group1_params = {'n':40, 'mean':10, 'sd':2}
group2_params = {'n':20, 'mean':5, 'sd':2}
group1 = group1_params['mean'] + np.random.randn(group1_params['n']) *
group1_params['sd']
group2 = group2_params['mean'] + np.random.randn(group2_params['n']) *
group2_params['sd']

y = np.hstack([group1, group2])
x = pd.DataFrame({'Intercept':np.ones(len(y)),
'Contrast':np.hstack([np.ones(group1_params['n']), -1*np.ones(group2_params['n'])])})

run_regression_simulation(x, y)
```

```
betas: [7.98765299 2.16002722]
beta1 + beta2: 10.147680205027868
beta1 - beta2: 5.827625771635669
mean(group1): 10.147680205027864
mean(group2): 5.8276257716356685
mean(group1) - mean(group2): 4.320054433392196
mean(y): 8.707662060563798
```



Looks like the beta estimates are identical to the previous simulation. This demonstrates that we *do not* need to adjust the weights of the number of ones and zeros to sum to zero. This is because the beta is estimating the average distance from the mean, which is invariant to group sizes.

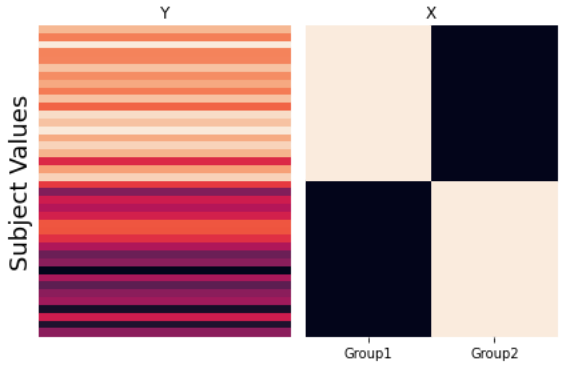
Independent-Samples T-Test - Group Intercepts

The third way to calculate an independent samples t-test using a regression is to split the intercept into two separate binary regressors with each reflecting the membership of each group. There is no need to include an intercept as it is simply a linear combination of the other two regressors.

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

```
group1_params = {'n':20, 'mean':10, 'sd':2}
group2_params = {'n':20, 'mean':5, 'sd':2}
group1 = group1_params['mean'] + np.random.randn(group1_params['n']) *
group1_params['sd']
group2 = group2_params['mean'] + np.random.randn(group2_params['n']) *
group2_params['sd']

y = np.hstack([group1, group2])
x = pd.DataFrame({'Group1':np.hstack([np.ones(len(group1)), np.zeros(len(group2))]),
'Group2':np.hstack([np.zeros(len(group1)), np.ones(len(group2))])})

run_regression_simulation(x, y)
```

```
betas: [9.41009741 4.34939829]
beta1 + beta2: 13.759495698990353
beta1 - beta2: 5.060699128393246
mean(group1): 9.410097413691801
mean(group2): 4.3493982852985535
mean(group1) - mean(group2): 5.060699128393248
mean(y): 6.879747849495177
```



This model is obviously separately estimating the means of each group, but how do we know if the difference is significant? Any ideas?

Just like the single subject regression models, we would need to calculate a contrast, which would simply be $c = [1 - 1]$.

All three of these different approaches will yield identical results when performing a hypothesis test, but each is computing the t-test slightly differently.

Paired-Samples T-Test

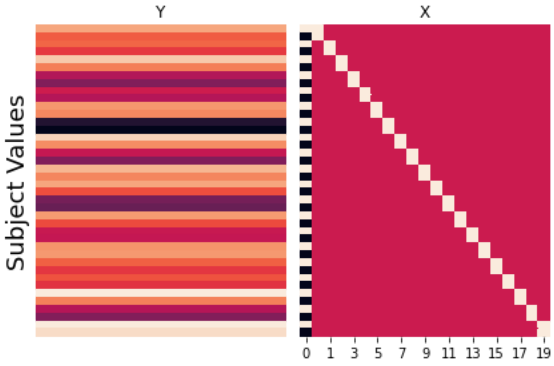
Now let's demonstrate that a paired-samples t-test can also be computed using a regression. Here, we will need to create a long format dataset, in which each subject s_i has two data points (one for each condition a and b). One regressor will compute the contrast between condition a and condition b. Just like before, we need to account for the mean, but instead of computing a grand mean for all of the data, we will separately model the mean of each participant by adding n more binary regressors where each subject is indicated in each regressor.

$$\begin{bmatrix} s_1 a \\ s_1 b \\ s_2 a \\ s_2 b \\ s_3 a \\ s_3 b \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

This simulation will be slightly more complicated as we will be adding subject level noise to each data point. In this simulation, we will assume that $\epsilon_i = \mathcal{N}(30, 10)$

```
a_params = {'mean':10, 'sd':2}
b_params = {'mean':5, 'sd':2}
sample_params = {'n':20, 'mean':30, 'sd':10}

y = []; x = []; sub_id = [];
for s in range(sample_params['n']):
    sub_mean = sample_params['mean'] + np.random.randn()*sample_params['sd']
    a = sub_mean + a_params['mean'] + np.random.randn() * a_params['sd']
    b = sub_mean + b_params['mean'] + np.random.randn() * b_params['sd']
    y.extend([a,b])
    x.extend([1, -1])
    sub_id.extend([s]*2)
y = np.array(y)

sub_means = pd.DataFrame([sub_id==x for x in np.unique(sub_id)]).T
sub_means = sub_means.replace({True:1,False:0})
X = pd.concat([pd.Series(x), sub_means], axis=1)

run_regression_simulation(X, y, paired=True)
```

```
betas: [38.56447613 35.08912938 42.0287038  22.7343386  28.18762639 42.88390721
  7.0837409  43.58511337 22.66066626 42.75568132 37.21848932 19.9169548
 36.84854258 30.06708857 44.8072347  34.76752398 34.36573871 42.14435925
 22.91057627 52.88492285]
contrast beta: 2.545226964528254
mean(subject betas): 36.62046768344662
mean(y): 36.620467683446634
mean(a): 39.16569464797489
mean(b): 34.07524071891838
mean(a-b): 5.090453929056513
sum(a_i-mean(y_i))/n: 2.5452269645282555
```



Okay, now let's try to make sense of all of these numbers. First, we now have n + 1 β's. β_0 corresponds to the between condition contrast. We will call this the *contrast* β. The rest of the β's model each subject's mean. We can see that the means of all of these subject β's corresponds to the overall mean of y.

Now what is the meaning of the contrast β?

We can see that it is not the average within subject difference between the two conditions as might be expected given a normal paired-samples t-test.

Instead, just like the independent samples t-test described above, the contrast value reflects the average deviation of a condition from each subject's individual mean.

$$\sum_{i=1}^{n} \frac{a_i - mean(y_i)}{n}$$

where n is the number of subjects, a is the condition being compared to b, and the $mean(y_i)$ is the subject's mean.

Linear and Quadratic contrasts

Hopefully, now you are starting to see that all of the different statistical tests you learned in intro stats (e.g., one-sample t-tests, two-sample t-tests, ANOVAs, and regressions) are really just a special case of the general linear model.

Contrasts allow us to flexibly test many different types of hypotheses within the regression framework. This allows us to test more complicated and precise hypotheses than might be possible than simply turning everything into a binary yes/no question (i.e., one sample t-test), or is condition a greater than condition b (i.e., two sample t-test). We've already explored how contrasts can be used to create independent and paired-samples t-tests in the above simulations. Here we will now provide examples of how to test more sophisticated hypotheses.

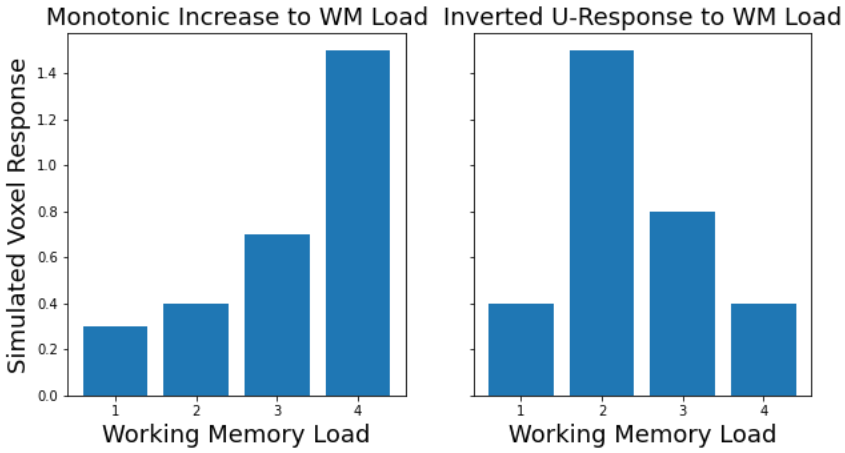
Suppose we manipulated the intensity of some type of experimental manipulation across many levels. For example, we increase the working memory load across 4 different levels. We might be interested in identifying regions that monotonically increase as a function of this manipulation. This would be virtually impossible to test using a paired contrast approach (e.g., t-tests, ANOVAs). Instead, we can simply specify a linear contrast by setting the contrast vector to linearly increase. This is as simple as `[0, 1, 2, 3]`. However, remember that contrasts need to sum to zero (except for the one-sample t-test case). So to make our contrast we can simply subtract the mean - `np.array([0, 1, 2, 3]) - np.mean((np.array([0, 1, 2, 3))`, which becomes $c_{linear} = [-1.5, -0.5, 0.5, 1.5]$.

Regions involved in working memory load might not have a linear increase, but instead might show an inverted u-shaped response, such that the region is not activated at small or high loads, but only at medium loads. To test this hypothesis, we would need to construct a quadratic contrast $c_{quadratic} = [-1, 1, 1, -1]$.

Let's explore this idea with a simple simulation.

```
# First let's make up some hypothetical data based on different types of response we
might expect to see.
sim1 = np.array([.3, .4, .7, 1.5])
sim2 = np.array([.4, 1.5, .8, .4])
x = [1,2,3,4]

# Now let's plot our simulated data
f,a = plt.subplots(ncols=2, sharey=True, figsize=(10, 5))
a[0].bar(x, sim1)
a[1].bar(x, sim2)
a[0].set_ylabel('Simulated Voxel Response', fontsize=18)
a[0].set_xlabel('Working Memory Load', fontsize=18)
a[1].set_xlabel('Working Memory Load', fontsize=18)
a[0].set_title('Monotonic Increase to WM Load', fontsize=18)
a[1].set_title('Inverted U-Response to WM Load', fontsize=18)
```

```
Text(0.5, 1.0, 'Inverted U-Response to WM Load')
```



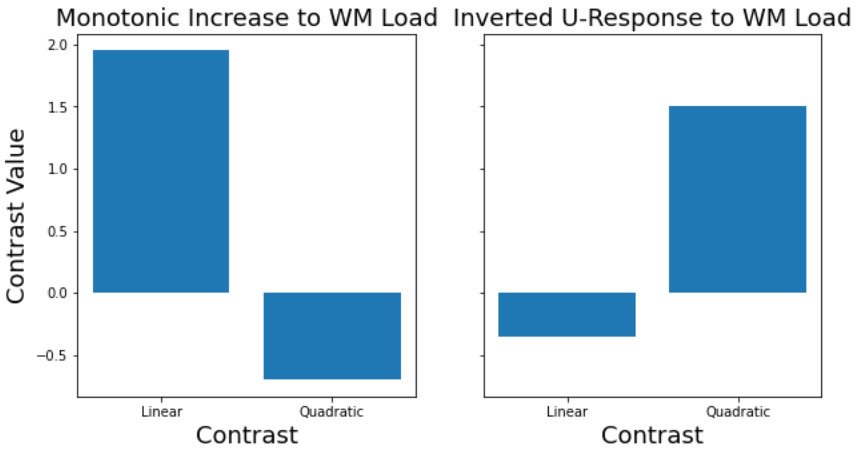
See how the data appear to have a linear and quadratic response to working memory load?

Now let's create some contrasts and see how a linear or quadratic contrast might be able to detect these different predicted responses.

```
# First let's create some contrast codes.
linear_contrast = np.array([-1.5, -.5, .5, 1.5])
quadratic_contrast = np.array([-1, 1, 1, -1])

print(f'Linear Contrast: {linear_contrast}')
print(f'Quadratic Contrast: {quadratic_contrast}')

# Now let's test our contrasts on each dataset.
sim1_linear = np.dot(sim1, linear_contrast)
sim1_quad = np.dot(sim1, quadratic_contrast)
sim2_linear = np.dot(sim2, linear_contrast)
sim2_quad = np.dot(sim2, quadratic_contrast)

# Now plot the contrast results
f,a = plt.subplots(ncols=2, sharey=True, figsize=(10,5))
a[0].bar(['Linear','Quadratic'], [sim1_linear, sim1_quad])
a[1].bar(['Linear','Quadratic'], [sim2_linear, sim2_quad])
a[0].set_ylabel('Contrast Value', fontsize=18)
a[0].set_xlabel('Contrast', fontsize=18)
a[1].set_xlabel('Contrast', fontsize=18)
a[0].set_title('Monotonic Increase to WM Load', fontsize=18)
a[1].set_title('Inverted U-Response to WM Load', fontsize=18)
```

```
Linear Contrast: [-1.5 -0.5  0.5  1.5]
Quadratic Contrast: [-1  1  1 -1]
```

```
Text(0.5, 1.0, 'Inverted U-Response to WM Load')
```



As you can see, the linear contrast is sensitive to detecting responses that monotonically increase, while the quadratic contrast is more sensitive to responses that show an inverted u-response. Both of these are also signed, so they could also detect responses in the opposite direction.

If we were to apply this to real brain data, we could now find regions that show a linear or quadratic responses to an experimental manipulation across the whole brain. We would then test the null hypothesis that there is no group effect of a linear or quadratic contrast at the second level.

Hopefully, this is starting you a sense of the power of contrasts to flexibly test any hypothesis that you can imagine.

Exercises

1. Which regions are more involved with visual compared to auditory sensory processing?

- Create a contrast to test this hypothesis
- run a group level t-test
- plot the results
- write the file to your output folder.

2. Which regions are more involved in processing numbers compared to words?

- Create a contrast to test this hypothesis
- run a group level t-test
- plot the results
- write the file to your output folder.

3. Are there gender differences?

In this exercise, create a two sample design matrix comparing men and women on arithmetic vs reading.

You will first have to figure out the subjects gender using the using the `participants.tsv` file.

- Create a contrast to test this hypothesis
- run a group level t-test
- plot the results
- write the file to your output folder.

```
from bids import BIDSLayout, BIDSValidator

data_dir = '../data/localizer'
layout = BIDSLayout(data_dir, derivatives=False)

meta_data = pd.read_csv(layout.get(suffix='participants', extension='.tsv')[0],
sep='\t')
meta_data.head()
```

Thresholding Group Analyses

Written by Luke Chang

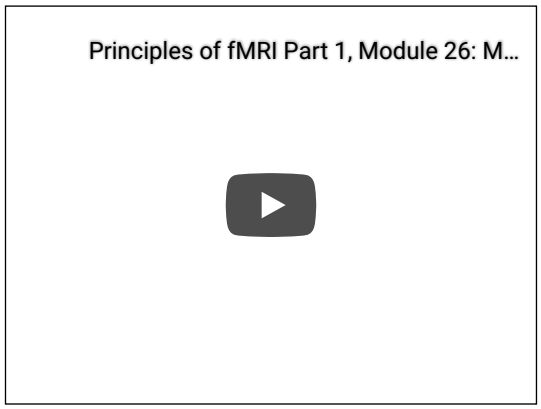
Now that we have learned how to estimate a single-subject model, create contrasts, and run a group-level analysis, the next important topic to cover is how we can threshold these group maps. This is not as straightforward as it might seem as we need to be able to correct for multiple comparisons.

In this tutorial, we will cover how we go from modeling brain responses in each voxel for a single participant to making inferences about the group. We will cover the following topics:

- Issues with correcting for multiple comparisons
- Family Wise Error Rate
- Bonferroni Correction
- False Discovery Rate

Let's get started by watching an overview of multiple comparisons by Martin Lindquist.

```
from IPython.display import YouTubeVideo

YouTubeVideo('AalIM9-5-Pk')
```



The primary goal in fMRI data analysis is to make inferences about how the brain processes information. These inferences can be in the form of predictions, but most often we are testing hypotheses about whether a particular region of the brain is involved in a specific type of process. This requires rejecting a H_0 hypothesis (i.e., that there is no effect). Null hypothesis testing is traditionally performed by specifying contrasts between different conditions of an experimental design and assessing if these differences between conditions are reliably present across many participants. There are two main types of errors in null-hypothesis testing.

Type I error

- H_0 is true, but we mistakenly reject it (i.e., False Positive)
- This is controlled by significance level α.

Type II error

- H_0 is false, but we fail to reject it (False Negative)

The probability that a hypothesis test will correctly reject a false null hypothesis is described as the *power* of the test.

Hypothesis testing in fMRI is complicated by the fact that we are running many tests across each voxel in the brain (hundreds of thousands of tests). Selecting an appropriate threshold requires finding a balance between sensitivity (i.e., true positive rate) and specificity (i.e., false negative rate). There are two main approaches to correcting for multiple tests in fMRI data analysis.

Familywise Error Rate (FWER) attempts to control the probability of finding *any* false positives. Mathematically, FWER can be defined as the probability P of observing any false positive $FWER = P(FalsePositives \geq 1)$.

While, **False Discovery Rate** (FDR) attempts to control the proportion of false positives among rejected tests. Formally, this is the expected proportion of false positive to the observed number of significant tests $FDR = E(\frac{FalsePositives}{SignificantTests})$.

This should probably be no surprise to anyone, but fMRI studies are expensive and inherently underpowered. Here is a simulation by Jeannette Mumford to show approximately how many participants you would need to achieve 80% power assuming a specific effect size in your contrast.

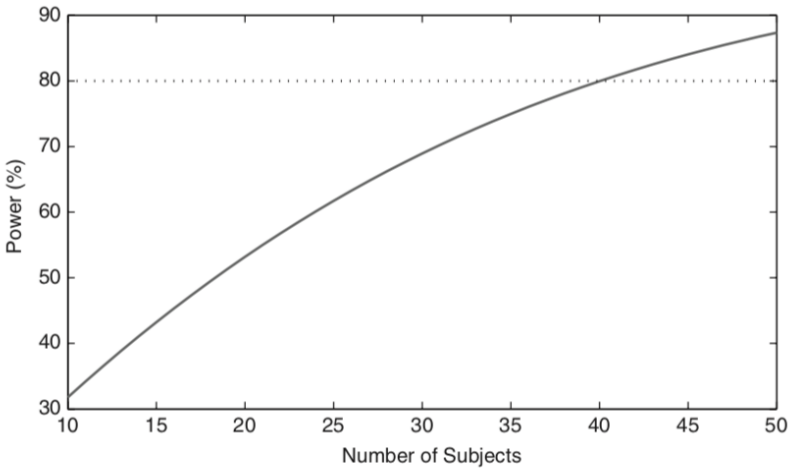


Figure 7.7. Power curve. The curve was generated using an estimated mean effect of 0.8% signal change units with standard deviation of 2% signal change units and a type I error rate of 0.05 using Equation 7.2. Since the graph crosses 80% between 40 and 41 subjects, a sample size of 41 will yield at least 80% power.

Simulations

Let's explore the concept of false positives to get an intuition about what the overall goals and issues are in controlling for multiple tests.

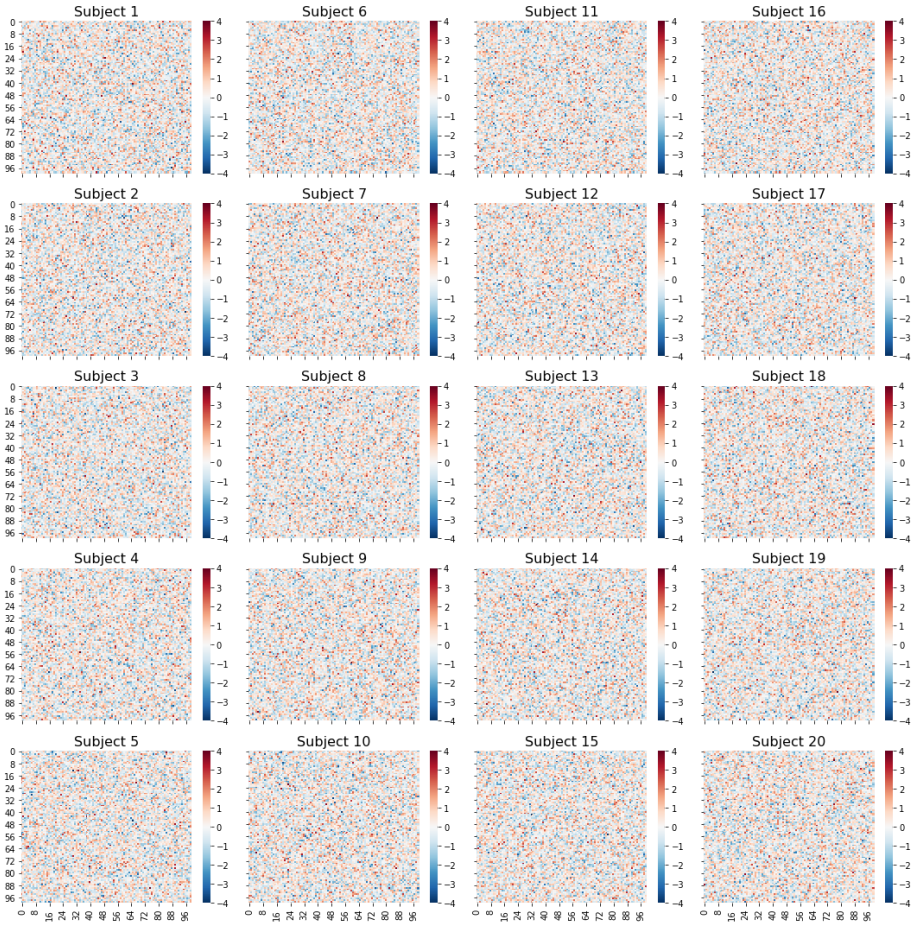
Let's load the modules we need for this tutorial. We will be using the SimulateGrid class which contains everything we need to run all of the simulations.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.data import Brain_Data
from nltools.simulator import SimulateGrid
```

```
/Users/lukechang/anaconda3/lib/python3.7/site-
packages/sklearn/utils/deprecation.py:144: FutureWarning: The
sklearn.linear_model.base module is  deprecated in version 0.22 and will be removed
in version 0.24. The corresponding classes / functions should instead be imported
from sklearn.linear_model. Anything that cannot be imported from sklearn.linear_model
is now part of the private API.
  warnings.warn(message, FutureWarning)
```

Okay, let's get started and generate 100 x 100 voxels from $\mathcal{N}(0, 1)$ distribution for 20 independent participants.

```
simulation = SimulateGrid(grid_width=100, n_subjects=20)

f,a = plt.subplots(nrows=5, ncols=4, figsize=(15,15), sharex=True, sharey=True)
counter = 0
for col in range(4):
    for row in range(5):
        sns.heatmap(simulation.data[:, :, counter], ax=a[row, col], cmap='RdBu_r',
vmin=-4, vmax=4)
        a[row,col].set_title(f'Subject {counter+1}', fontsize=16)
        counter += 1
plt.tight_layout()
```



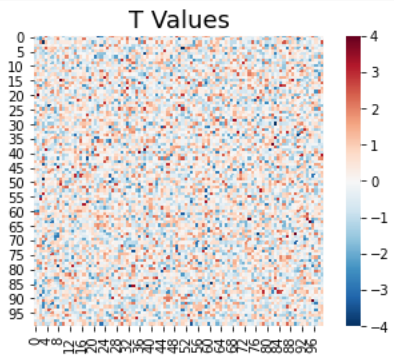
Each subject's simulated data is on a 100 x 100 grid. Think of this as a slice from their brain, where each pixel corresponds to the same spatial location across all participants. We have generated random noise separately for each subject. We have not added any true signal in this simulation yet.

This figure is simply to highlight that we are working with 20 independent subjects. In the rest of the plots, we will be working with a single grid that aggregates the results across participants.

Now we are going to start running some simulations to get a sense of the number of false positives we might expect to observe with this data. We will now run an independent one-sample t-test on every pixel in the grid across all 20 participants.

```
simulation.fit()

sns.heatmap(simulation.t_values, square=True, cmap='RdBu_r', vmin=-4, vmax=4)
plt.title("T Values", fontsize=18)
```

```
Text(0.5, 1.0, 'T Values')
```



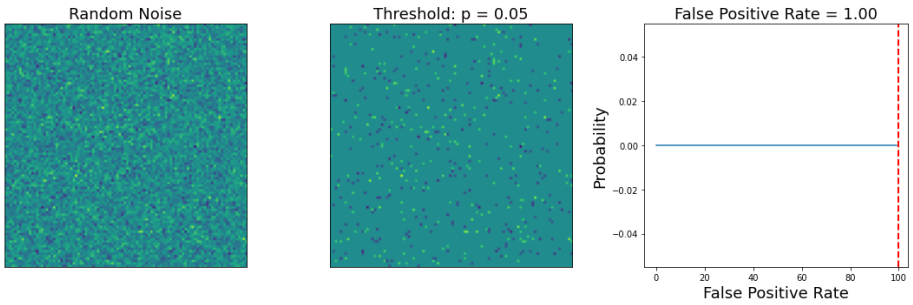
Even though there was no signal in this simulation, you can see that there are a number of pixels in the grid that exceed a t-value above 2 and below -2, which is the approximate cutoff for $p < 0.05$. These are all false positives.

Now let's apply a threshold. We can specify thresholds at a specific t-value using the `threshold_type='t'`. Alternatively, we can specify a specific p-value using the `threshold_type='p'`. To calculate the number of false positives, we can simply count the number of tests that exceed this threshold.

If we run this simulation again 100 times, we can estimate the false positive rate, which is the average number of false positives over all 100 simulations.

Let's see what this looks like for a threshold of p < 0.05.

```
threshold = .05
simulation = SimulateGrid(grid_width=100, n_subjects=20)
simulation.plot_grid_simulation(threshold=threshold, threshold_type='p',
n_simulations=100)
```

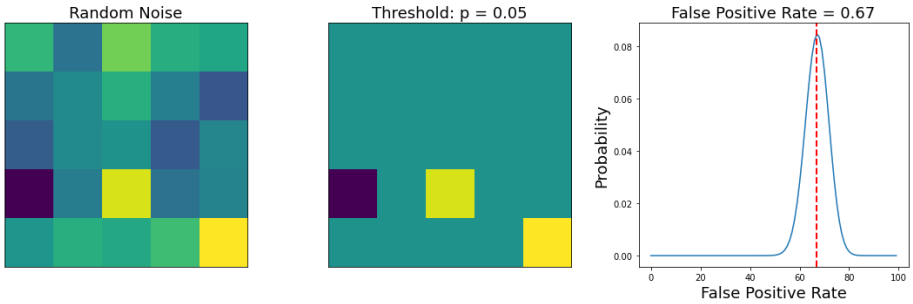


The left panel is the average over all of the participants. The middle panel show voxels that exceed the statistical threshold. The right panel is the overall false-positive rate across the 100 simulations.

In this simulation, a threshold of p < 0.05 results in observing at least one voxels that is a false positive across every one of our 100 simulations.

What if we looked at a fewer number of voxels? How would this change our false positive rate?

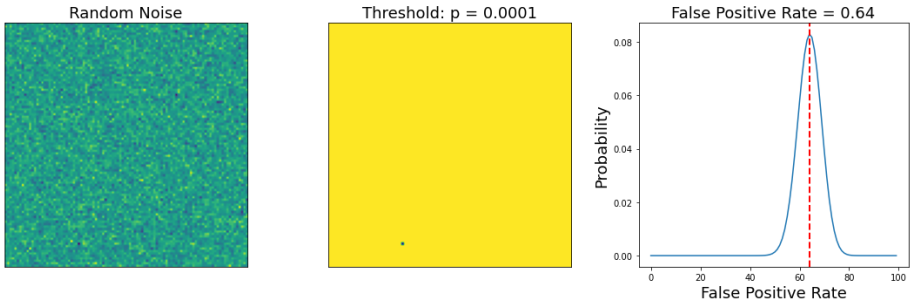
```
threshold = .05
simulation = SimulateGrid(grid_width=5, n_subjects=20)
simulation.plot_grid_simulation(threshold=threshold, threshold_type='p',
n_simulations=100)
```



This simulation shows that examining fewer numbers of voxels will yield considerably less false positives. One common approach to controlling for multiple tests involves only looking for voxels within a specific region of interest (e.g., small volume correction), or looking at average activation within a larger region (e.g., ROI based analyses).

What about if we increase the threshold on our original 100 x 100 grid?

```
threshold = .0001
simulation = SimulateGrid(grid_width=100, n_subjects=20)
simulation.plot_grid_simulation(threshold=threshold, threshold_type='p',
n_simulations=100)
```



You can see that this dramatically decreases the number of false positives to the point that some of the simulations no longer contain any false positives.

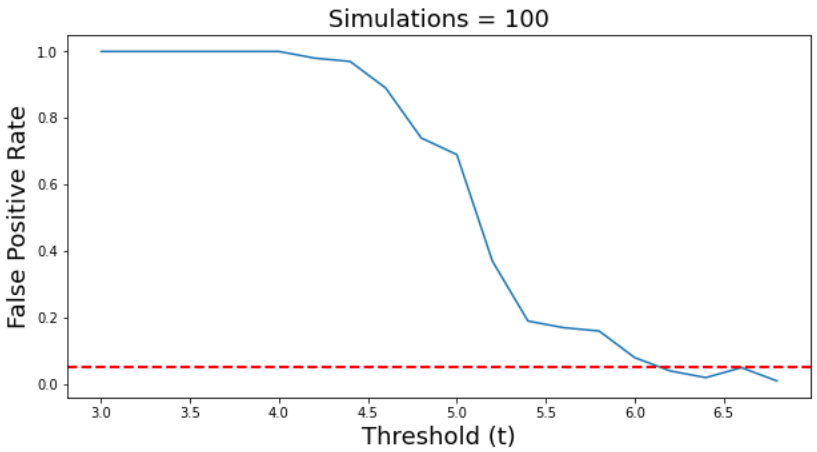
What is the optimal threshold that will give us an $\alpha = 0.05$?

To calculate this, we will run 100 simulations at different threshold levels to find the threshold that leads to a false positive rate that is lower than our alpha value.

We could search over t-values, or p-values. Let's explore t-values first.

```
alpha = 0.05
n_simulations = 100
x = np.arange(3, 7, .2)

sim_all = []
for p in x:
    sim = SimulateGrid(grid_width=100, n_subjects=20)
    sim.run_multiple_simulations(threshold=p, threshold_type='t',
n_simulations=n_simulations)
    sim_all.append(sim.fpr)

f,a = plt.subplots(ncols=1, figsize=(10, 5))
a.plot(x, np.array(sim_all))
a.set_ylabel('False Positive Rate', fontsize=18)
a.set_xlabel('Threshold (t)', fontsize=18)
a.set_title(f'Simulations = {n_simulations}', fontsize=18)
a.axhline(y=alpha, color='r', linestyle='dashed', linewidth=2)
```

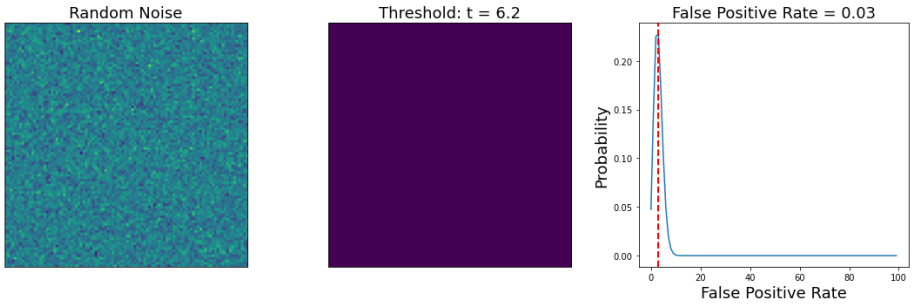
```
<matplotlib.lines.Line2D at 0x7feb9fc76290>
```



As you can see, the false positive rate is close to our alpha starting at a threshold of about 6.2. This means that when we test a hypothesis over 10,000 independent voxels, we can be confident that we will only observe false positives in approximately 5 out of 100 experiments. This means that we are effectively controlling the family wise error rate (FWER).

Let's use that threshold for our simulation again.

```
simulation = SimulateGrid(grid_width=100, n_subjects=20)
simulation.plot_grid_simulation(threshold=6.2, threshold_type='t', n_simulations=100)
```

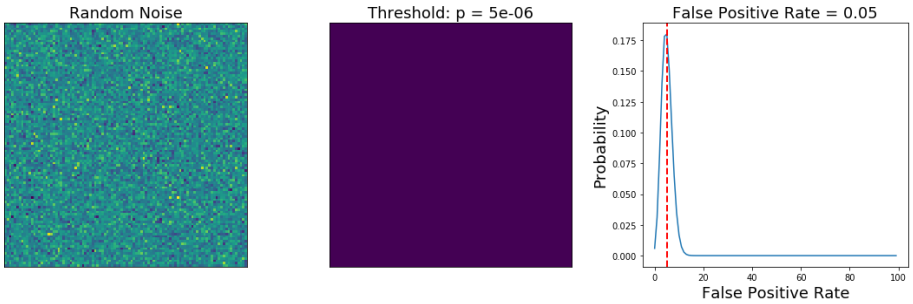


Notice, that we are now observing a false positive rate of approximately .05, though this number will slightly change each time you run the simulation.

Another way to find the threshold that controls FWER is to divide the alpha by the number of independent tests across voxels. This is called the **bonferroni correction**.

$bonferroni = \frac{\alpha}{M}$, where M is the number of voxels.

```
grid_width = 100
threshold = 0.05/(grid_width**2)
simulation = SimulateGrid(grid_width=grid_width, n_subjects=20)
simulation.plot_grid_simulation(threshold=threshold, threshold_type='p',
n_simulations=100)
```



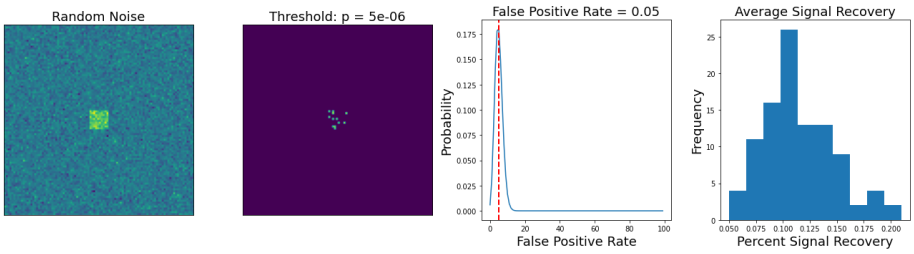
This seems like a great way to ensure that we minimize our false positives.

Now what happens when start adding signal to our simulation?

We will represent signal in a smaller square in the middle of the simulation. The width of the square can be changed using the `signal_width` parameter. The amplitude of this signal is controlled by the `signal_amplitude` parameter.

Let's see how well the bonferroni threshold performs when we add 100 voxels of signal.

```
grid_width = 100
threshold = .05/(grid_width**2)
signal_width = 10
signal_amplitude = 1

simulation = SimulateGrid(signal_amplitude=signal_amplitude, signal_width=10,
grid_width=grid_width, n_subjects=20)
simulation.plot_grid_simulation(threshold=threshold, threshold_type='p',
n_simulations=100)
```



Here we show how many voxels were identified using the bonferroni correction.

In the left panel is the average data across all 20 participants. The second panel, shows the voxels that exceed the statistical threshold. The third panel shows the false positive rate, and the 4th panel furthest on the right shows the average signal recovery (how many voxels survived within the true signal square across all 100 simulations.

We can see that we have an effective false positive rate approximately equal to our alpha threshold. However, our threshold is so high, that we can barely detect any true signal with this amplitude. In fact, we are only recovering about 12% of the voxels that should have signal.

This simulation highlights the main issue with using bonferroni correction in practice. The threshold is so conservative that the magnitude of an effect needs to be unreasonably large to survive correction over hundreds of thousands of voxels.

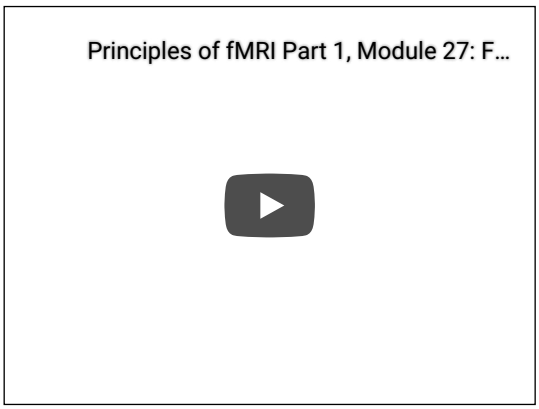
Family Wise Error Rate

At this point you may be wondering if it even makes sense to assume that each test is independent. It seems reasonable to expect some degree of spatial correlation in our data. Our simulation is a good example of this as we have a square that contains signal across contiguous voxels. In practice, most of our functional neuroanatomy that we are investigating is larger than a single voxel and our spatial smoothing preprocessing step increase the spatial correlation.

It can be shown that the Bonferroni correction is overally conservative in the presence of spatial dependence and results in a decreased power to detect voxels that are truly active.

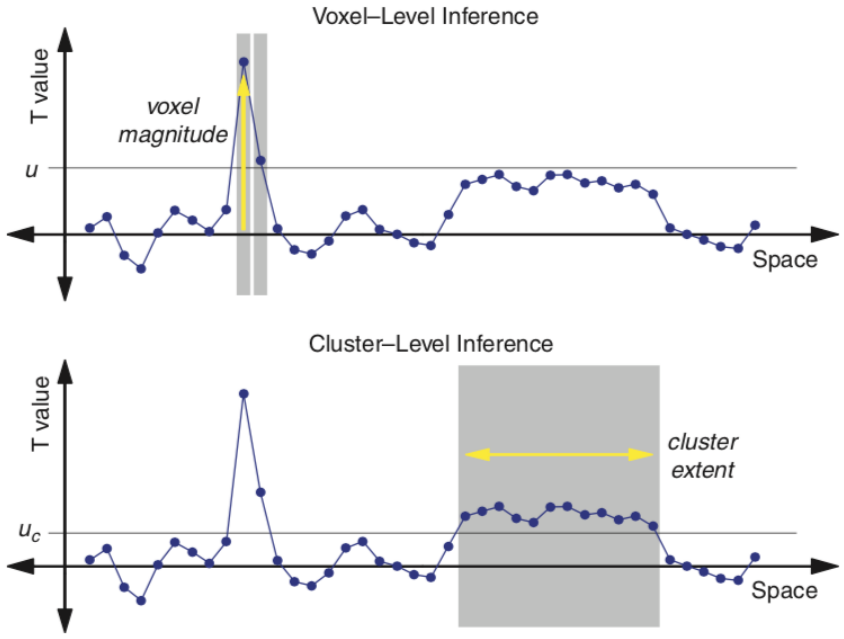
Let's watch a video by Martin Lindquist to learn more about different ways to control for the Family Wise Error Rate.

```
YouTubeVideo('MxQeEdVNihg')
```

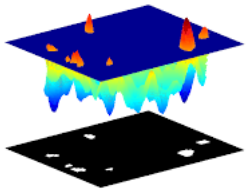


Cluster Extent

Another approach to controlling the FWER is called cluster correction, or cluster extent. In this approach, the goal is to identify a threshold such that the maximum statistic exceeds it at a specified alpha. The distribution of the maximum statistic can be approximated using Gaussian Random Field Theory (RFT), which attempts to account for the spatial dependence of the data.



This requires specifying an initial threshold to determine the *Euler Characteristic* or the number of blobs minus the number of holes in the thresholded image. The number of voxels in the blob and the overall smoothness can be used to calculate something called *resels* or resolution elements and can be effectively thought of as the spatial units that need to be controlled for using FWER. We won't be going into too much detail with this approach as the mathematical details are somewhat complicated. In practice, if the image is smooth and the number of subjects is high enough (around 20), cluster correction seems to provide control closer to the true false positive rate than Bonferroni correction. Though we won't be spending time simulating this today, I encourage you to check out this Python simulation by Matthew Brett and this chapter for an introduction to random field theory.



Cluster extent thresholding has recently become somewhat controversial due to several high profile papers that have found that it appears to lead to an inflated false positive rate in practice (see Ekland et al., 2017). A recent paper by Woo et al. 2014 has shown that a liberal initial threshold (i.e. higher than $p < 0.001$) will inflate the number of false positives above the nominal level of 5%. There is no optimal way to select the initial threshold and often slight changes will give very different results. Furthermore, this approach does not appear to work equally well across all types of findings. For example, this approach can work well with some amounts of smoothing results that have a particular spatial extent, but not equally well for all types of signals. In other words, it seems potentially problematic to assume that spatial smoothness is constant over the brain and also that it is adequately represented using a Gaussian distribution. Finally, it is important to note that this approach only allows us to make inferences for the entire cluster. We can say that there is some voxel in the cluster that is significant, but we can't really pinpoint which voxels within the cluster may be driving the effect.

This is one of the more popular ways to control for multiple comparisons as it is particularly sensitive when there is a weak, but spatially contiguous signal. However, in practice, we don't recommend using this approach as it has a lot of assumptions that are rarely met and the spatial inference is fairly weak.

There are several other popular FWER approaches to correcting for multiple tests that try to address these issues.

Threshold Free Cluster Extent

One interesting solution to the issue of finding an initial threshold seems to be addressed by the threshold free cluster enhancement method presented in Smith & Nichols, 2009. In this approach, the authors propose a way to combine cluster extent and voxel height into a single metric that does not require specifying a specific initial threshold. It essentially involves calculating the integral of the overall product of a signal intensity and spatial extent over multiple thresholds. It has been shown to perform particularly well when combined with non-parameteric resampling approaches such as randomise in FSL. This method is implemented in FSL and also in Matlab by Mark Thornton. For more details about this approach check out this blog post by Mark Thornton, this video by Jeanette Mumford, and the original technical report.

Parametric simulations

One approach to estimating the inherent smoothness in the data, or it's spatial autocorrelation, is using parametric simulations. This was the approach originally adopted in AFNI's AlphaSim/3DClustSim. After it was demonstrated that real fMRI data was not adequately modeled by a standard Gaussian distribution, the AFNI group quickly updated their software and implemented a range of different algorithms in their 3DClustSim tool. See this paper for an overview of these changes.

Nonparametric approaches

As an alternative to RFT, nonparametric methods use the data themselves to find the appropriate distribution. These methods can provide substantial improvements in power and validity, particularly with small sample sizes, so we recommend these in general over cluster extent. These tests can verify the validity of the less computationally expensive parametric approaches. However, it is important to note that this is much more computationally expensive as 5-10k permutations need to be run at every voxel. The FSL tool randomise is probably the current gold standard and there are versions that run on GPUs, such as BROCCOLI to speed up the computation time.

Here we will run a simulation using a one-sample permutation test (i.e., sign test) on our data. We will make the grid much smaller to speed up the simulation and will decrease the number of simulations by an order of magnitude, but you will see that it is still very slow (5,000 permutations times 9 voxels times 10 simulations). This approach makes no distributional assumptions, but still requires correcting for multiple tests using either FWER or FDR approaches. Don't worry about running this cell if it is taking too long.

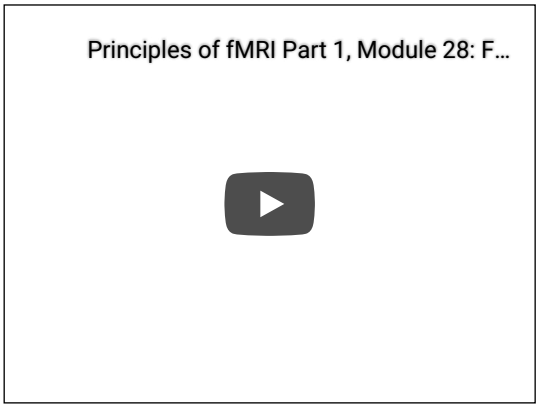
```
grid_width = 3
threshold = .05
signal_amplitude = 1
simulation = SimulateGrid(signal_amplitude=signal_amplitude, signal_width=2,
grid_width=grid_width, n_subjects=20)
simulation.t_values, simulation.p_values =
simulation._run_permutation(simulation.data)
simulation.isfit = True
simulation.threshold_simulation(threshold, 'p')
simulation.plot_grid_simulation(threshold=threshold, threshold_type='p',
n_simulations=10)
```

False Discovery Rate (FDR)

You may be wondering why we need to control for *any* false positive when testing across hundreds of thousands of voxels. Surely a few are okay as long as they don't overwhelm the true signal.

Let's learn about the False Discovery Rate (FDR) from another video by Martin Lindquist.

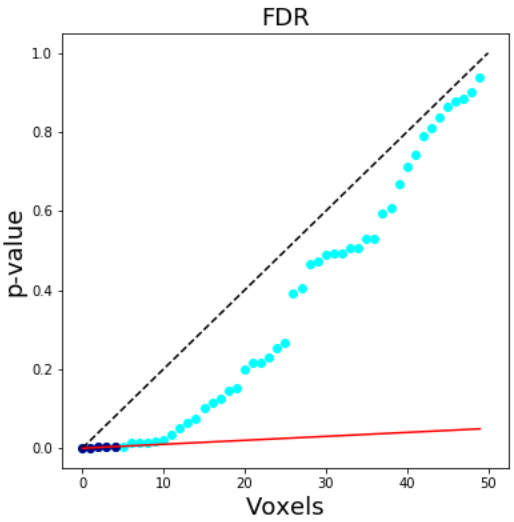
```
YouTubeVideo('W9ogBO4GEzA')
```



The *false discovery rate* (FDR) is a more recent development in multiple testing correction originally described by Benjamini & Hochberg, 1995. While FWER is the probability of any false positives occurring in a family of tests, the FDR is the expected proportion of false positives among significant tests.

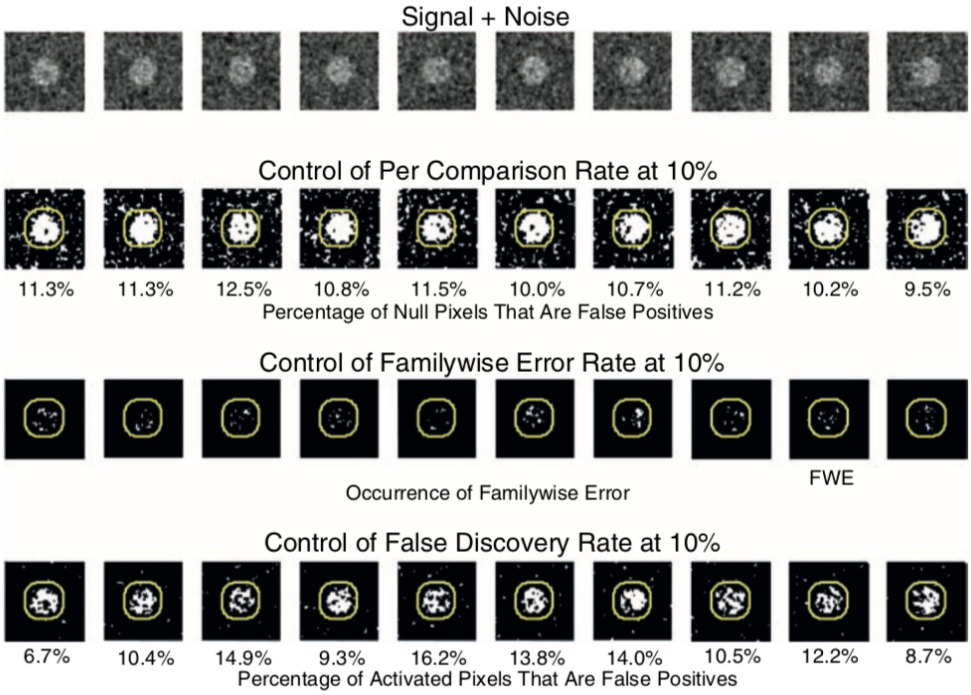
The FDR is fairly straightforward to calculate.

1. We select a desired limit q on FDR, which is the proportion of false positives we are okay with observing (e.g., 5/100 tests or 0.05).
2. We rank all of the p-values over all the voxels from the smallest to largest.
3. We find the threshold r such that $p \leq i/m * q$
4. We reject any H_0 that is lower than r.



In a brain map, this means that we expect approximately 95% of the voxels reported at q < .05 FDR-corrected to be true activations (note we use q instead of p). The FDR procedure adaptively identifies a threshold based on the overall signal across all voxels. Larger signals results in lower thresholds. Importantly, if all of the null hypotheses are true, then the FDR will be equivalent to the FWER. This means that any FWER procedure will *also* control the FDR. For these reasons, any procedure which controls the FDR is necessarily less stringent than a FWER controlling procedure, which leads to an overall increased power. Another nice feature of FDR, is that it operates on p-values instead of test statistics, which means it can be applied to most statistical tests.

This figure is taken from Poldrack, Mumford, & Nichols (2011) and compares different procedures to control for multiple tests.

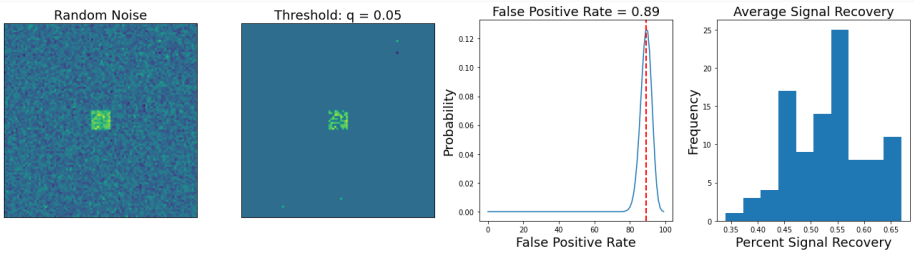


For a more indepth overview of FDR, see this tutorial by Matthew Brett.

Let's now try to apply FDR to our own simulations. All we need to do is add a `correction='fdr'` flag to our simulation plot. We need to make sure that the `threshold=0.05` to use the correct q.

```
grid_width = 100
threshold = .05
signal_amplitude = 1
simulation = SimulateGrid(signal_amplitude=signal_amplitude, signal_width=10,
grid_width=grid_width, n_subjects=20)
simulation.plot_grid_simulation(threshold=threshold, threshold_type='q',
n_simulations=100, correction='fdr')
print(f'FDR q < 0.05 corresponds to p-value of {simulation.corrected_threshold}')
```

```
FDR q < 0.05 corresponds to p-value of 0.00030256885357668906
```

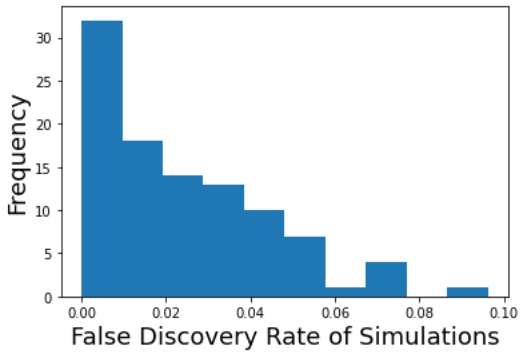


Okay, using FDR of q < 0.05 for our simulation identifies a p-value threshold of p < 0.00034. This is more liberal than the bonferroni threshold of p < 0.000005 and allows us to recover much more signal as a consequence. You can see that at this threshold there are more false positives, which leads to a much higher overall false positive rate. Remember, this metric is only used for calculating the family wise error rate and indicates the presence of *any* false positive across each of our 100 simulations.

To calculate the empirical false discovery rate, we need to calculate the percent of any activated voxels that were false positives.

```
plt.hist(simulation.multiple_fdr)
plt.ylabel('Frequency', fontsize=18)
plt.xlabel('False Discovery Rate', fontsize=18)
plt.xlabel('False Discovery Rate of Simulations', fontsize=18)
```

```
Text(0.5, 0, 'False Discovery Rate of Simulations')
```



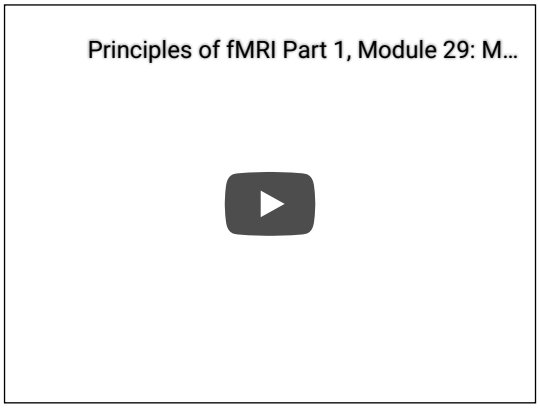
In our 100 simulations, the majority had a false discovery rate below our q < 0.05.

Thresholding Brain Maps

In the remainder of the tutorial, we will move from simulation to playing with real data.

Let's watch another video by Tor Wager on how multiple comparison approaches are used in practice, highlighting some of the pitfalls with some of the different approaches.

```
YouTubeVideo('N7Iittt8HrU')
```

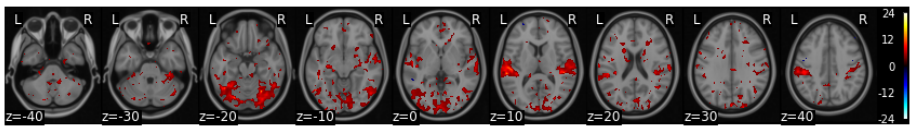


We will be exploring two simple and fast ways to threshold your group analyses.

First, we will simply threshold based on selecting an arbitrary statistical threshold. The values are completely arbitrary, but it is common to start with something like p < .001. We call this *uncorrected* because this is simply the threshold for any voxel as we are not controlling for multiple tests.

```
con1_name = 'horizontal_checkerboard'
data_dir = '../data/localizer'
con1_file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep','*', 'func',
f'sub*_{con1_name}*nii.gz'))
con1_file_list.sort()
con1_dat = Brain_Data(con1_file_list)
con1_stats = con1_dat.ttest(threshold_dict={'unc':.001})

con1_stats['thr_t'].plot()
```

```
threshold is ignored for simple axial plots
```



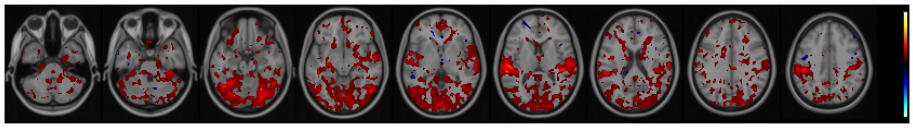
We see some significant activations in visual cortex, but we also see strong t-tests in the auditory cortex.

Why do you think this is?

We can also easily run FDR correction by changing the inputs of the `threshold_dict`. We will be using a q value of 0.05 to control our false discovery rate.

```
con1_stats = con1_dat.ttest(threshold_dict={'fdr':.05})
con1_stats['thr_t'].plot()
```

```
threshold is ignored for simple axial plots
```

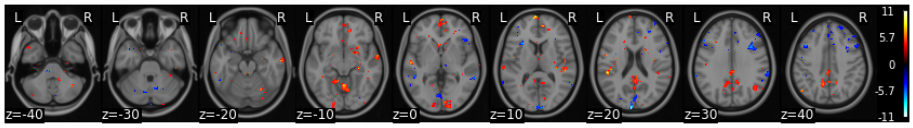


You can see that at least for this particular contrast, the FDR threshold appears to be more liberal than p < 0.001 uncorrected.

Let's look at another contrast between vertical and horizontal checkerboards.

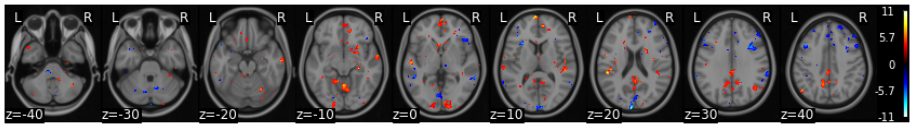
```
con2_name = 'vertical_checkerboard'
con2_file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep','*', 'func',
f'sub*_{con2_name}*nii.gz'))
con2_file_list.sort()
con2_dat = Brain_Data(con2_file_list)

con1_v_con2 = con1_dat-con2_dat

con1_v_con2_stats = con1_v_con2.ttest(threshold_dict={'unc':.001})
con1_v_con2_stats['thr_t'].plot()
```

```
threshold is ignored for simple axial plots
```




```
con1_v_con2_stats = con1_v_con2.ttest(threshold_dict={'fdr':.05})
con1_v_con2_stats['thr_t'].plot()
```

```
threshold is ignored for simple axial plots
```



Looks like there are some significant differences that survive in early visual cortex and also in other regions of the brain.

This concludes are very quick overview to performing univariate analyses in fMRI data analysis.

We will continue to add more advanced tutorials to the dartbrains.org website. Stay tuned!

Exercises

Exercise 1. Bonferroni Correction Simulation

Using the Grid Simulation code above, try to find how much larger the signal needs to be using a Bonferroni Correction until we can recover 100% of the true signal, while controlling a family wise error false-positive rate of p < 0.05.

Exercise 2. Which regions are more involved with visual compared to auditory sensory processing?

- run a group level t-test and threshold using an uncorrected voxel-wise threshold of p < 0.05, p < 0.005, and p < 0.001.
- plot each of the results
- write each file to your output folder.

Exercise 3. Which regions are more involved in processing numbers compared to words?

- run a group level t-test, using a correcte FDR threshold of q < 0.05.
- plot the results
- write the file to your output folder.

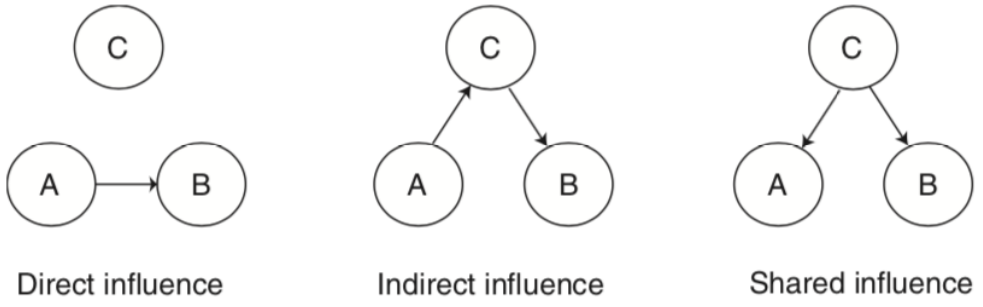
Connectivity

Written by Luke Chang

So far, we have primarily been focusing on analyses related to task evoked brain activity. However, an entirely different way to study the brain is to characterize how it is intrinsically connected. There are many different ways to study functional connectivity.

The primary division is studying how brain regions are *structurally* connected. In animal studies this might involve directly tracing bundles of neurons that are connected to other neurons. Diffusion imaging is a common way in which we can map how bundles of white matter are connected to each region, based on the direction in which water diffuses along white matter tracks. There are many different techniques such as fractional ansiotropy and probablistic tractography. We will not be discussing structural connectivity in this course.

An alternative approach to studying connectivity is to examine how brain regions covary with each other in time. This is referred to as *functional connectivity*, but it is better to think about it as temporal covariation between regions as this does not necessarily imply that two regions are directly communication with each other.



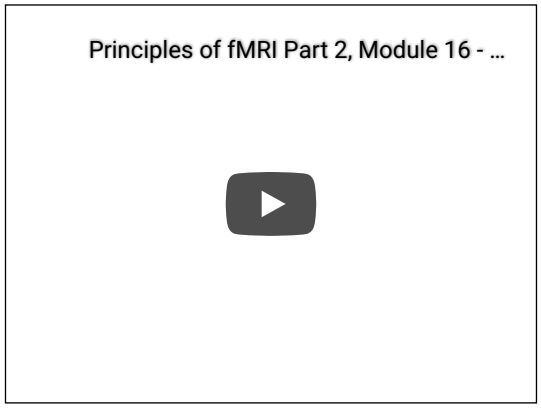
For example, regions can *directly* influence each other, or they can *indirectly* influence each other via a mediating region, or they can be affected similarly by a *shared influence*. These types of figures are often called *graphs*. These types of *graphical* models can be *directed* or *undirected*. Directed graphs imply a causal relationship, where one region A directly influence another region B. Directed graphs or *causal models* are typically described as *effective connectivity*, while undirected graphs in which the relationship is presumed to be bidirectional are what we typically describe as *functional connectivity*.

In this tutorial, we will work through examples on:

- Seed-based functional connectivity
- Psychophysiological interactions
- Principal Components Analysis
- Graph Theory

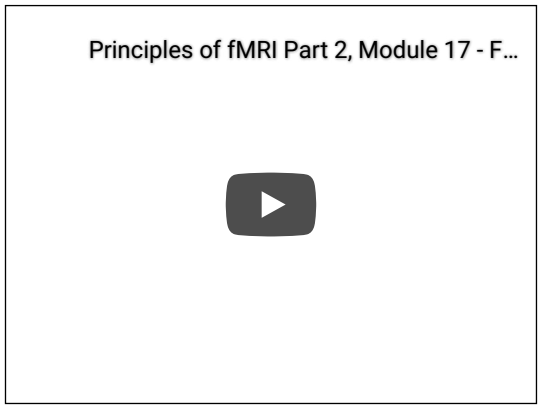
Let's start by watching a short overview of connectivity by Martin Lindquist.

```
from IPython.display import YouTubeVideo

YouTubeVideo('J0KX_rW0hmc')
```



Now, let's dive in a little bit deeper into the specific details of functional connectivity.

```
YouTubeVideo('OVAQujut_1o')
```



Functional Connectivity

Seed Voxel Correlations

One relatively simple way to calculate functional connectivity is to compute the temporal correlation between two regions of interest (ROIs). Typically, this is done by extracting the temporal response from a *seed voxel* or the average response within a *seed region*. Then this time course is regressed against all other voxels in the brain to produce a whole brain map of anywhere that shares a similar time course to the seed.

Let's try it ourselves with an example subject from the Pinel Localizer dataset. First, let's import the modules we need for this tutorial and set our paths.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.data import Brain_Data, Design_Matrix, Adjacency
from nltools.mask import expand_mask, roi_to_brain
from nltools.stats import zscore, fdr, one_sample_permutation
from nltools.file_reader import onsets_to_dm
from nltools.plotting import component_viewer
from scipy.stats import binom, ttest_1samp
from sklearn.metrics import pairwise_distances
from copy import deepcopy
import networkx as nx
from nilearn.plotting import plot_stat_map, view_img_on_surf
from bids import BIDSLayout, BIDSValidator
import nibabel as nib

base_dir = '/Users/lukechang/Dropbox/Dartbrains'
data_dir = os.path.join(base_dir, 'data', 'Localizer')
layout = BIDSLayout(data_dir, derivatives=True)
```

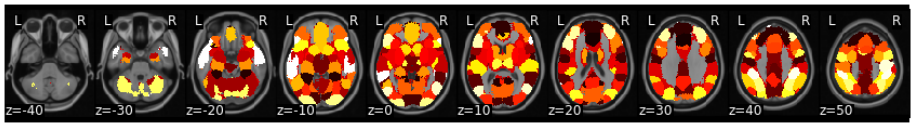
Now let's load an example participant's preprocessed functional data.

```
sub = 'S01'
fwhm=6

data = Brain_Data(layout.get(subject=sub, task='localizer', scope='derivatives',
suffix='bold', extension='nii.gz', return_type='file')[0])
smoothed = data.smooth(fwhm=fwhm)
```

Next we need to pick an ROI. Pretty much any type of ROI will work.

In this example, we will be using a whole brain parcellation based on similar patterns of coactivation across over 10,000 published studies available in neurosynth (see this paper for more details). We will be using a parcellation of 50 different functionally similar ROIs.

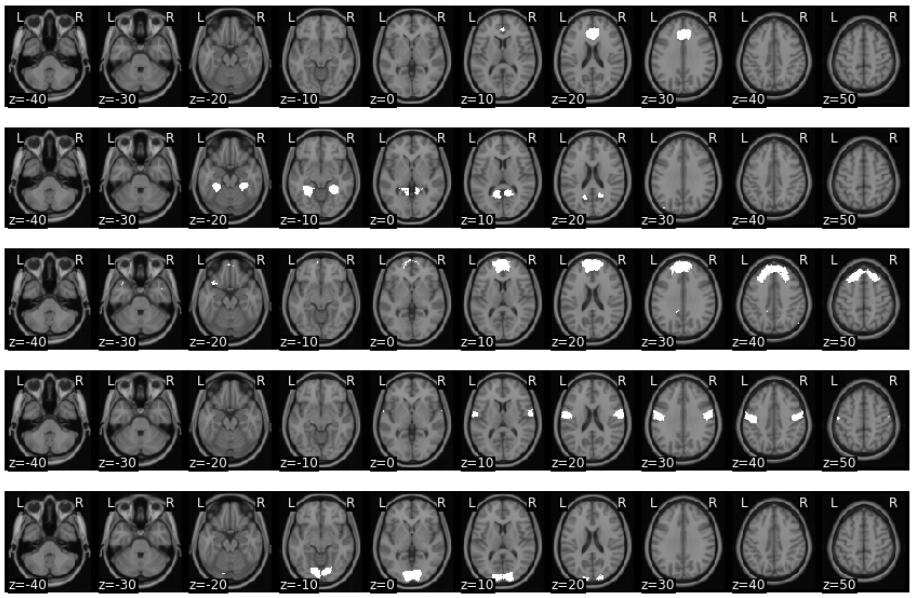
```
mask = Brain_Data('https://neurovault.org/media/images/8423/k50_2mm.nii.gz')

mask.plot()
```



Each ROI in this parcellation has its own unique number. We can expand this so that each ROI becomes its own binary mask using `nltools.mask.expand_mask`.

Let's plot the first 5 masks.

```
mask_x = expand_mask(mask)

f = mask_x[0:5].plot()
```

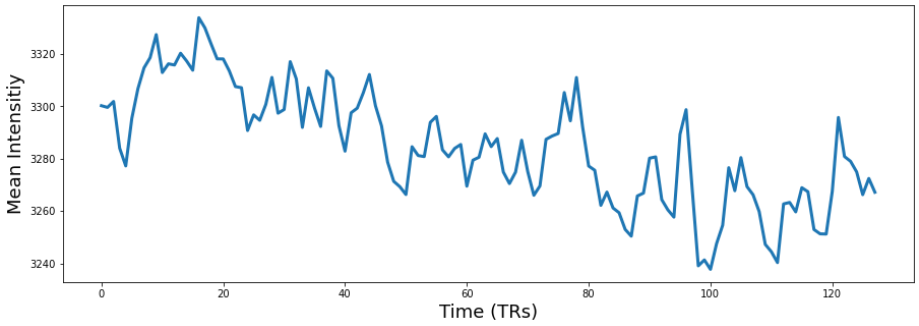


To use any mask we just need to index it by the correct label.

Let's start by using the vmPFC mask (ROI=32) to use as a seed in a functional connectivity analysis.

```
vmpfc = smoothed.extract_roi(mask=mask_x[32])

plt.figure(figsize=(15,5))
plt.plot(vmpfc, linewidth=3)
plt.ylabel('Mean Intensitiy', fontsize=18)
plt.xlabel('Time (TRs)', fontsize=18)
```

```
Text(0.5, 0, 'Time (TRs)')
```



Okay, now let's build our regression design matrix to perform the whole-brain functional connectivity analysis.

The goal is to find which regions in the brain have a similar time course to the vmPFC, controlling for all of our covariates (i.e., nuisance regressors).

Functional connectivity analyses are particularly sensitive to artifacts that might induce a temporal relationship, particularly head motion (See this article by Jonathan Power for more details). This means that we will need to use slightly different steps to preprocess data for this type of analyis then a typical event related mass univariate analysis.

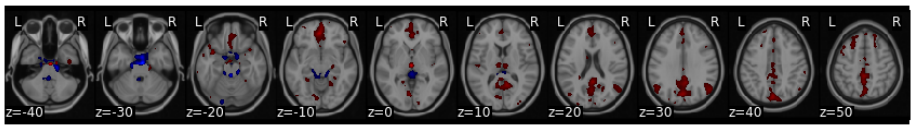
We are going to remove the mean from our vmPFC signal. We are also going to include the average activity in CSF as an additional nuisance regressor to remove physiological artifacts. Finally, we will be including our 24 motion covariates as well as linear and quadratic trends. We need to be a little careful about filtering as the normal high pass filter for an event related design might be too short and will remove potential signals of interest.

Resting state researchers also often remove the global signal, which can reduce physiological and motion related artifacts and also increase the likelihood of observing negative relationships with your seed regressor (i.e., anticorrelated). This procedure has remained quite controversial in practice (see here here, here, and here for a more in depth discussion). We think that in general including covariates like CSF should be sufficient. It is also common to additionally include covariates from white matter masks, and also multiple principal components of this signal rather than just the mean (see more details about compcorr.

Overall, this code should seem very familiar as it is pretty much the same procedure we used in the single subject GLM tutorial. However, instead of modeling the task design, we are interested in calculating the functional connectivity with the vmPFC.

```
tr = layout.get_tr()
fwhm = 6
n_tr = len(data)

def make_motion_covariates(mc, tr):
    z_mc = zscore(mc)
    all_mc = pd.concat([z_mc, z_mc**2, z_mc.diff(), z_mc.diff()**2], axis=1)
    all_mc.fillna(value=0, inplace=True)
    return Design_Matrix(all_mc, sampling_freq=1/tr)


vmpfc = zscore(pd.DataFrame(vmpfc, columns=['vmpfc']))

csf_mask = Brain_Data(os.path.join(base_dir, 'masks', 'csf.nii.gz'))
csf = zscore(pd.DataFrame(smoothed.extract_roi(mask=csf_mask).T, columns=['csf']))

spikes = smoothed.find_spikes(global_spike_cutoff=3, diff_spike_cutoff=3)
covariates = pd.read_csv(layout.get(subject=sub, scope='derivatives', extension='.tsv')
[0].path, sep='\t')
mc = covariates[['trans_x','trans_y','trans_z','rot_x', 'rot_y', 'rot_z']]
mc_cov = make_motion_covariates(mc, tr)
dm = Design_Matrix(pd.concat([vmpfc, csf, mc_cov, spikes.drop(labels='TR', axis=1)],
axis=1), sampling_freq=1/tr)
dm = dm.add_poly(order=2, include_lower=True)

smoothed.X = dm
stats = smoothed.regress()

vmpfc_conn = stats['beta'][0]
```

```
vmpfc_conn.threshold(upper=25, lower=-25).plot()
```



Notice how this analysis identifies the default network? This analysis is very similar to the original papers that identified the default mode network using resting state data.

For an actual analysis, we would need to repeat this procedure over all of the participants in our sample and then perform a second level group analysis to identify which voxels are consistently coactive with the vmPFC. We will explore group level analyses in the exercises.

Psychophysiological Interactions

Suppose we were interested in seeing if the vmPFC was connected to other regions differently when performing a finger tapping task compared to all other conditions. To compute this analysis, we will need to create a new design matrix that combines the motor regressors and then calculates an interaction term between the seed region activity (e.g., vmpfc) and the condition of interest (e.g., motor).

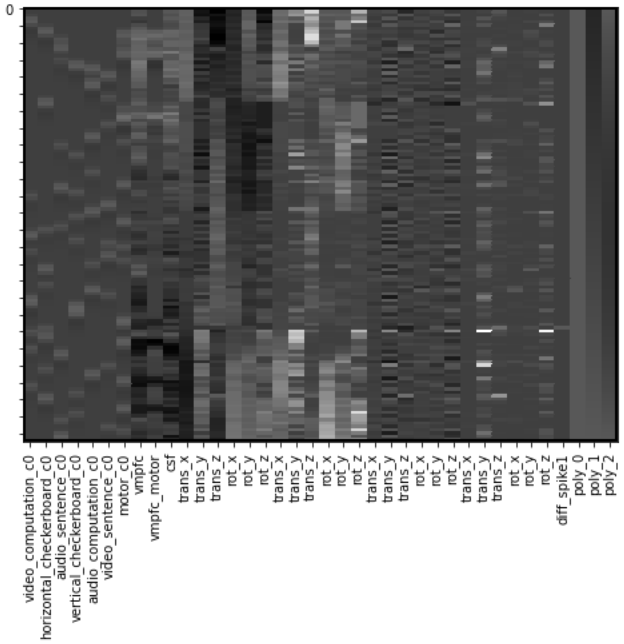
This type of analysis called, *psychophysiological* interactions was originally proposed by Friston et al., 1997. For a more hands on and practical discussion read this paper and watch this video by Jeanette Mumford and a follow up video) of a more generalized method.

```
nib.load(layout.get(subject='S01', scope='raw', suffix='bold')[0].path)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
~/anaconda3/lib/python3.7/site-packages/nibabel/loadsave.py in load(filename,
**kwargs)
     41     try:
---> 42         stat_result = os.stat(filename)
     43     except OSError:

FileNotFoundError: [Errno 2] No such file or directory:
'/Users/lukechang/Dropbox/Dartbrains/data/Localizer/sub-S01/func/sub-S01_task-
localizer_bold.nii.gz'

During handling of the above exception, another exception occurred:

FileNotFoundError                         Traceback (most recent call last)
<ipython-input-25-6aac70d09b2b> in <module>
----> 1 nib.load(layout.get(subject='S01', scope='raw', suffix='bold')[0].path)

~/anaconda3/lib/python3.7/site-packages/nibabel/loadsave.py in load(filename,
**kwargs)
     42         stat_result = os.stat(filename)
     43     except OSError:
---> 44         raise FileNotFoundError("No such file or no access: '%s'" % filename)
     45     if stat_result.st_size <= 0:
     46         raise ImageFileError("Empty file: '%s'" % filename)

FileNotFoundError: No such file or no access:
'/Users/lukechang/Dropbox/Dartbrains/data/Localizer/sub-S01/func/sub-S01_task-
localizer_bold.nii.gz'
```

```
def load_bids_events(layout, subject):
    '''Create a design_matrix instance from BIDS event file'''

    tr = layout.get_tr()
    n_tr = nib.load(layout.get(subject=subject, scope='raw', suffix='bold')
[0].path).shape[-1]

    onsets = pd.read_csv(layout.get(subject=subject, suffix='events')[0].path,
sep='\t')
    onsets.columns = ['Onset', 'Duration', 'Stim']
    return onsets_to_dm(onsets, sampling_freq=1/tr, run_length=n_tr)

dm = load_bids_events(layout, 'S01')
motor_variables = ['video_left_hand','audio_left_hand', 'video_right_hand',
'audio_right_hand']
ppi_dm = dm.drop(motor_variables, axis=1)
ppi_dm['motor'] = pd.Series(dm.loc[:, motor_variables].sum(axis=1))
ppi_dm_conv = ppi_dm.convolve()
ppi_dm_conv['vmpfc'] = vmpfc
ppi_dm_conv['vmpfc_motor'] = ppi_dm_conv['vmpfc']*ppi_dm_conv['motor_c0']
dm = Design_Matrix(pd.concat([ppi_dm_conv, csf, mc_cov, spikes.drop(labels='TR',
axis=1)], axis=1), sampling_freq=1/tr)
dm = dm.add_poly(order=2, include_lower=True)

dm.heatmap()
```

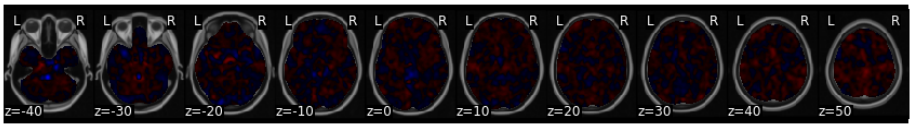
```
/Users/lukechang/anaconda3/lib/python3.7/site-packages/nltools-0.4.2-
py3.7.egg/nltools/file_reader.py:141: UserWarning: Computed onsets for
video_right_hand are inconsistent with expected values. Please manually verify the
outputted Design_Matrix!
  f"Computed onsets for {data.Stim.unique()[i]} are inconsistent with expected
values. Please manually verify the outputted Design_Matrix!"
/Users/lukechang/anaconda3/lib/python3.7/site-packages/nltools-0.4.2-
py3.7.egg/nltools/file_reader.py:141: UserWarning: Computed onsets for
video_left_hand are inconsistent with expected values. Please manually verify the
outputted Design_Matrix!
  f"Computed onsets for {data.Stim.unique()[i]} are inconsistent with expected
values. Please manually verify the outputted Design_Matrix!"
/Users/lukechang/anaconda3/lib/python3.7/site-packages/nltools-0.4.2-
py3.7.egg/nltools/file_reader.py:141: UserWarning: Computed onsets for
audio_computation are inconsistent with expected values. Please manually verify the
outputted Design_Matrix!
  f"Computed onsets for {data.Stim.unique()[i]} are inconsistent with expected
values. Please manually verify the outputted Design_Matrix!"
```



Okay, now we are ready to run the regression analysis and inspect the interaction term to find regions where the connectivity profile changes as a function of the motor task.

We will run the regression and smooth all of the images, and then examine the beta image for the PPI interaction term.

```
smoothed.X = dm
ppi_stats = smoothed.regress()

vmpfc_motor_ppi = ppi_stats['beta'][int(np.where(smoothed.X.columns=='vmpfc_motor')[0]
[0])]

vmpfc_motor_ppi.plot()
```



This analysis tells us which regions are more functionally connected with the vmPFC during the motor conditions relative to the rest of experiment.

We can make a thresholded interactive plot to interrogate these results, but it looks like it identifies the ACC/pre-SMA.

```
vmpfc_motor_ppi.iplot()
```

Dynamic Connectivity

All of the methods we have discussed so far assume that the relationship between two regions is stationary - or remains constant over the entire dataset. However, it is possible that voxels are connected to other voxels at specific points in time, but then change how they are connected when they are computing a different function or in different psychological state.

Time-varying connectivity is beyond the scope of the current tutorial, but we encourage you to watch this video from Principles of fMRI for more details

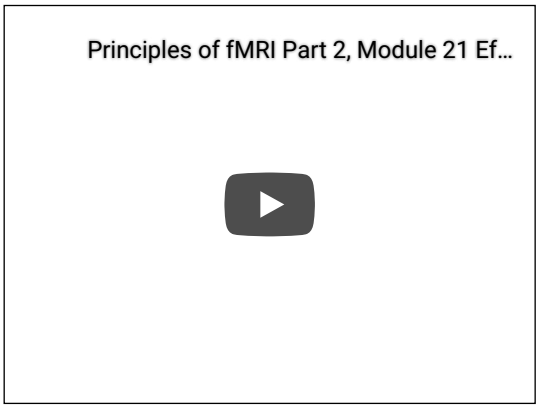
Effective Connectivity

Effective connectivity refers to the degree that one brain region has a directed influence on another region. This approach requires making a number of assumptions about the data and requires testing how well a particular model describes the data. Typically, most researchers will create a model of a small number of nodes and compare different models to each other. This is because the overall model fit is typically in itself uninterpretable and because formulating large models can be quite difficult and computationally expensive. The number of connections can be calculated as:

$connections = \frac{n(n-1)}{2}$, where n is the total number of nodes.

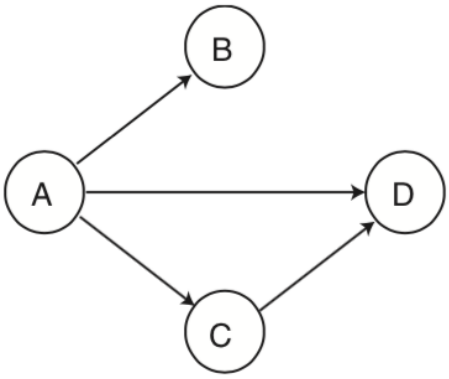
Let's watch a short video by Martin Lindquist that provides an overview to different approaches to effectivity connectivity.

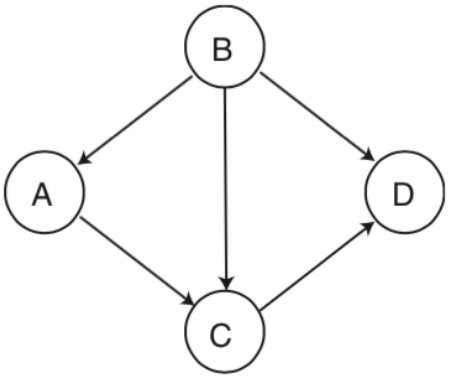
```
YouTubeVideo('gv5ENgW0bbs')
```



Structural Equation Modeling

Structural equation modeling (SEM) is one early technique that was used to model the causal relationship between multiple nodes. SEM requires specifying a causal relationship between nodes in terms of a set of linear equations. The parameters of this system of equations reflects the connectivity matrix. Users are expected to formulate their own hypothesized relationship between variables with a value of one when there is an expected relationship, and zero when there is no relationship. Then we estimate the parameters of the model and evaluate how well the model describes the observed data.





To \ From	A	B	C	D
A	0	0	0	0
B	1	0	0	0
C	1	0	0	0
D	1	0	1	0

To \ From	A	B	C	D
A	0	1	0	0
B	0	0	0	0
C	1	1	0	0
D	0	1	1	0

We will not be discussing this method in much detail. In practice, this method is more routinely used to examine how brain activations mediate relationships between other regions, or between different psychological constructs (e.g., X -> Z -> Y).

Here are a couple of videos specifically examining how to conduct mediation and moderation analyses from Principles of fMRI (Mediation and Moderation Part I, Mediation and Moderation Part II)

Granger Causality

Granger causality was originally developed in econometrics and is used to determine temporal causality. The idea is to quantify how past values of one brain region predict the current value of another brain region. This analysis can also be performed in the frequency domain using measures of coherence between two regions. In general, this technique is rarely used in fMRI data analysis as it requires making assumptions that all regions have the same hemodynamic response function (which does not seem to be true), and that the relationship is stationary, or not varying over time.

Here is a video from Principles of fMRI explaining Granger Causality in more detail.

Dynamic Causal Modeling

Dynamic Causal Modeling (DCM) is a method specifically developed for conducting causal analyses between regions of the brain for fMRI data. The key innovation is that the developers of this method have specified a generative model for how neuronal firing will be reflected in observed BOLD activity. This addresess one of the problems with SEM, which assumes that each ROI has the same hemodynamic response.

In practice, DCM is computationally expensive to estimate and researchers typically specify a couple small models and perform a model comparison (e.g., bayesian model comparison) to determine, which model best explains the data from a set of proposed models.

Here is a video from Principles of fMRI explaining Dynamic Causal Modeling in more detail.

Multivariate Decomposition

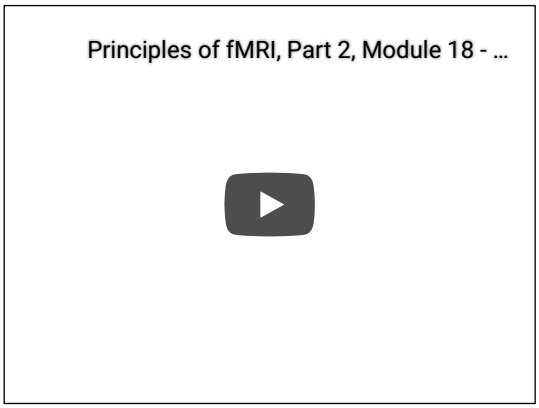
So far we have discussed functional connectivity in terms of pairs of regions. However, voxels are most likely not independent from each other and we may want to figure out some latent spatial components that are all functionally connected with each (i.e., covary similarly in time).

To do this type of analysis, we typically use what are called *multivariate decomposition* methods, which attempt to factorize a data set (i.e., time by voxels) into a lower dimensional set of components, where each has their own unique time course.

The most common decomposition methods or Principal Components Analysis (PCA) and Independent Components Analysis (ICA).

Let's watch a short video by Martin Lindquist to learn more about decomposition.

```
from IPython.display import YouTubeVideo

YouTubeVideo('Klp-8t5GLEg')
```



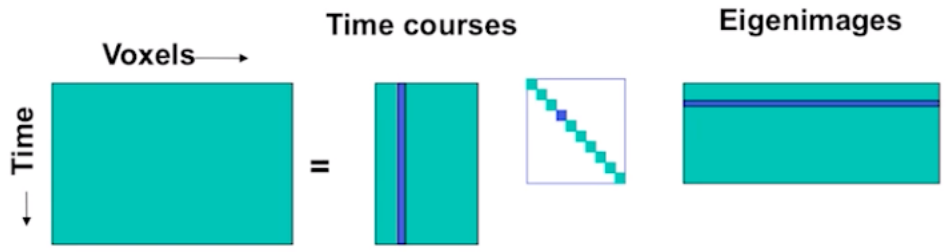
Principal Components Analysis

Principal Components Analysis (PCA) is a multivariate procedure that attempts to explain the variance-covariance structure of a high dimensional random vector. In this procedure, a set of correlated variables are transformed int a set of uncorrelated variables, ordered by the amount of variance in the data that they explain.

In fMRI, we use PCA to find spatial maps or *eigenimages* in the data. This is usually computed using Singular Value Decomposition (SVD). This operation is defined as:

$X = USV^T$, where $V^TV = I, U^TU = I$, and S is a diagonal matrix whose elements are called singular values.

In practice, V corresponds to the eigenimages or spatial components and U corresponds to the transformation matrix to convert the eigenimages into a timecourse. S reflects the amount of scaling for each component.



SVD is conceptually very similar to regression. We are trying to explain a matrix X as a linear combination of components. Each term in the equation reflects a unique (i.e., orthogonal) multivariate signal present in X. For example, the nth signal in X can be described by the dot product of a time course u_n and the spatial map Vn^T scaled by s_n.

$X = s_1u_1v_1^T + s_2u_2v_2^T + s_nu_nv_n^T$

Let's try running a PCA on our single subject data.

First, let's denoise our data using a GLM comprised only of nuisance regressors. We will then work with the *residual* of this model, or what remains of our data that was not explained by the denoising model. This is essentially identical to the vmPFC analysis, except that we will not be including any seed regressors. We will then be working with the residual of our regression, which is the remaining signal after removing any variance associated with our covariates.

```
csf_mask = Brain_Data(os.path.join(base_dir, 'masks', 'csf.nii.gz'))
csf = zscore(pd.DataFrame(smoothed.extract_roi(mask=csf_mask).T, columns=['csf']))

spikes = smoothed.find_spikes(global_spike_cutoff=3, diff_spike_cutoff=3)
covariates = pd.read_csv(layout.get(subject=sub, scope='derivatives', extension='.tsv')
[0].path, sep='\t')
mc = covariates[['trans_x','trans_y','trans_z','rot_x', 'rot_y', 'rot_z']]
mc_cov = make_motion_covariates(mc, tr)
dm = Design_Matrix(pd.concat([vmpfc, csf, mc_cov, spikes.drop(labels='TR', axis=1)],
axis=1), sampling_freq=1/tr)
dm = dm.add_poly(order=2, include_lower=True)

smoothed.X = dm
stats = smoothed.regress()

smoothed_denoised=stats['residual']
```

Now let's run a PCA on this participant's denoised data. To do this, we will use the `.decompose()` method from nltools. All we need to do is specify the algorithm we want to use, the dimension we want to reduce (i.e., time - 'images' or space 'voxels'), and the number of components to estimate. Usually, we will be looking at reducing space based on similarity in time, so we will set `axis='images'`.

```
n_components = 10

pca_stats_output = smoothed_denoised.decompose(algorithm='pca', axis='images',
n_components=n_components)
```

Now let's inspect the components with our interactive component viewer. Remember the ICA tutorial? Hopefully, you are now better able to understand everything.

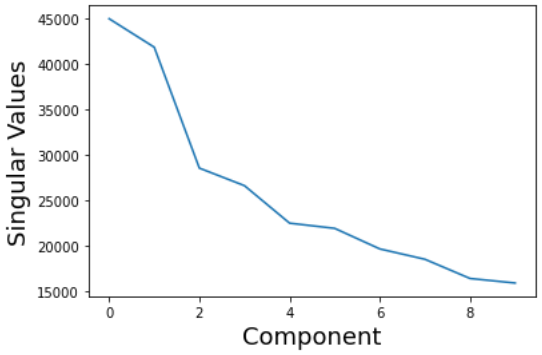
```
component_viewer(pca_stats_output, tr=layout.get_tr())
```

We can also examine the eigenvalues/singular values or scaling factor of each, which are the diagonals of S.

These values are stored in the `'decomposition_object'` of the stats_output and are in the variable called `.singular_values_`.

```
plt.plot(pca_stats_output['decomposition_object'].singular_values_)
plt.xlabel('Component', fontsize=18)
plt.ylabel('Singular Values', fontsize=18)
```

```
Text(0, 0.5, 'Singular Values')
```

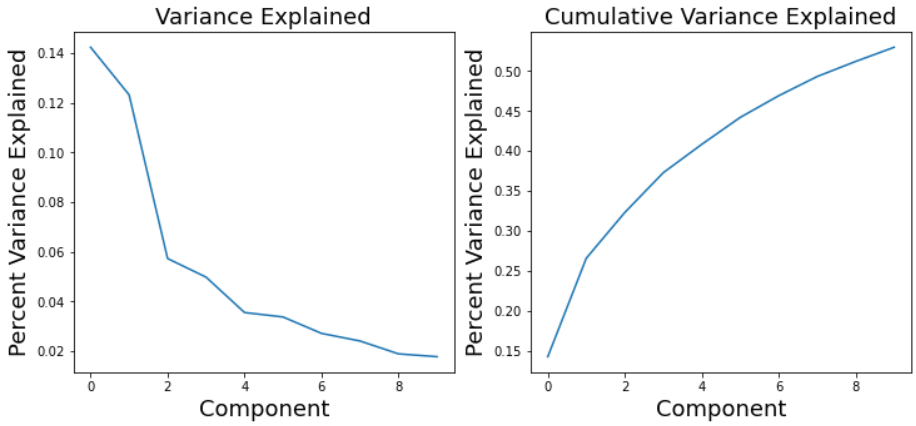


We can use these values to calculate the overall variance explained by each component. These values are stored in the `'decomposition_object'` of the stats_output and are in the variable called `.explained_variance_ratio_`.

These values can be used to create what is called a *scree* plot to figure out the percent variance of X explained by each component. Remember, in PCA, components are ordered by descending variance explained.

```
f,a = plt.subplots(ncols=2, figsize=(12, 5))
a[0].plot(pca_stats_output['decomposition_object'].explained_variance_ratio_)
a[0].set_ylabel('Percent Variance Explained', fontsize=18)
a[0].set_xlabel('Component', fontsize=18)
a[0].set_title('Variance Explained', fontsize=18)
a[1].plot(np.cumsum(pca_stats_output['decomposition_object'].explained_variance_ratio_)
)
a[1].set_ylabel('Percent Variance Explained', fontsize=18)
a[1].set_xlabel('Component', fontsize=18)
a[1].set_title('Cumulative Variance Explained', fontsize=18)
```

```
Text(0.5, 1.0, 'Cumulative Variance Explained')
```



Independent Components Analysis

Independent Components Analysis (ICA) is a method to blindly separate a source signal into spatially independent components. This approach assumes that the data consts of p spatially independent components, which are linearly mixed and spatially fixed. PCA assumes orthonormality constraint, while ICA only assumes independence.

$X = AS$, where A is the *mixing matrix* and S is the *source matrix*

In ICA we find an un-mixing matrix W, such that $Y = WX$ provides an approximation to S. To estimate the mixing matrix, ICA assumes that the sources are (1) linearly mixed, (2) the components are statistically independent, and (3) the components are non-Gaussian.

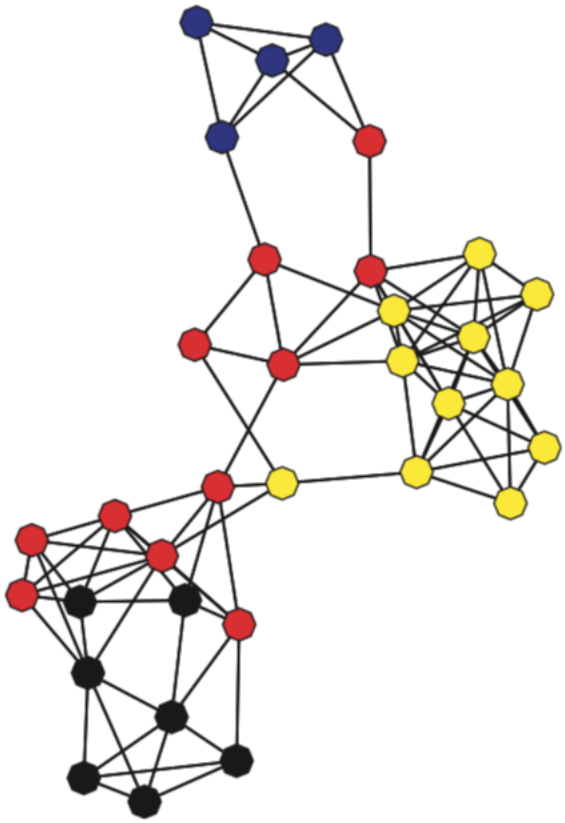
It is trivial to run ICA on our data as it only requires switching `algorithm='pca'` to `algorithm='ica'` when using the `decompose()` method.

We will experiment with this in our exercises.

Graph Theory

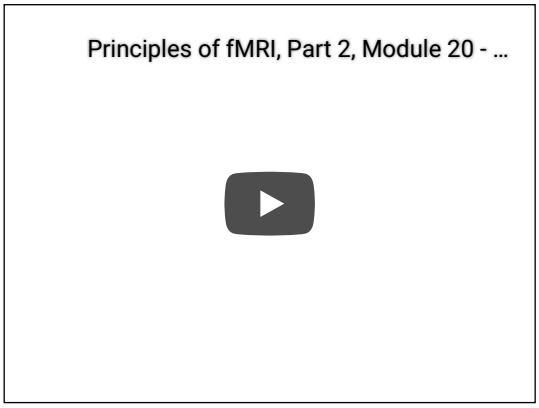
Similar to describing the structure of social networks, graph theory has also been used to characterize regions of the brain based on how they connected to other regions. Nodes in the network typically describe specific brain regions and edges represent the strength of the association between each edge. That is, the network can be represented as a graph of pairwise relationships between each region of the brain.

There are many different metrics of graphs that can be used to describe the overall efficiency of a network (e.g., small worldness), or how connected a region is to other regions (e.g., degree, centrality), or how long it would take to send information from one node to another node (e.g., path length, connectivity).



Let's watch a short video by Martin Lindquist providing a more in depth introduction to graph theory.

```
YouTubeVideo('v8ls5VED1ng')
```



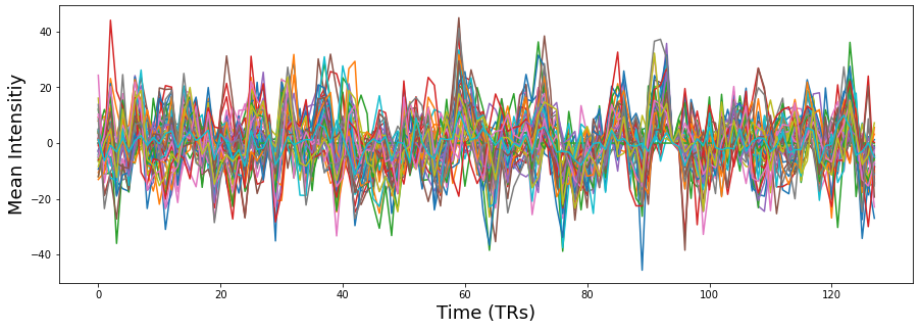
Suppose, we were interested in identifying which regions of the brain had the highest degree of centrality based on functional connectivity. There are many different ways to do this, but they all involve specifying a set of nodes (i.e., ROIs) and calculating the edges between each node. Finally, we would need to pick a centrality metric and calculate the overall level of centrality for each region.

Let's do this quickly building off of our seed-based functional connectivity analysis.

Similar, to the PCA example, let's work with the denoised data. First, let's extract the average time course within each ROI from our 50 parcels and plot the results.

```
rois = smoothed_denoised.extract_roi(mask=mask)

plt.figure(figsize=(15,5))
plt.plot(rois.T)
plt.ylabel('Mean Intensitiy', fontsize=18)
plt.xlabel('Time (TRs)', fontsize=18)
```

```
Text(0.5, 0, 'Time (TRs)')
```

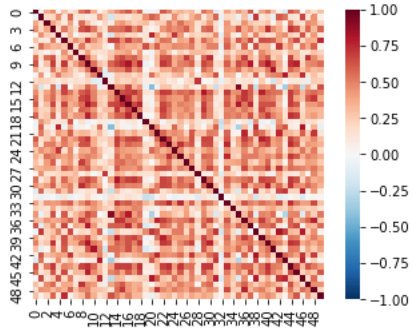


Now that we have specified our 50 nodes, we need to calculate the edges of the graph. We will be using pearson correlations. We will be using the `pairwise_distances` function from scikit-learn as it is much faster than most other correlation measures. We will then convert the distance metric into similarities by subtracting all of the values from 1.

Let's visualize the resulting correlation matrix as a heatmap using seaborn.

```
roi_corr = 1 - pairwise_distances(rois, metric='correlation')

sns.heatmap(roi_corr, square=True, vmin=-1, vmax=1, cmap='RdBu_r')
```

```
<AxesSubplot:>
```

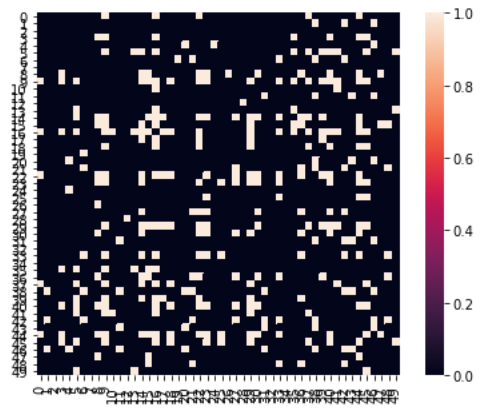


Now we need to convert this correlation matrix into a graph and calculate a centrality measure. We will use the `Adjacency` class from nltools as it has many functions that are useful for working with this type of data, including casting these type of matrices into networkx graph objects.

We will be using the networkx python toolbox to work with graphs and compute different metrics of the graph.

Let's calculate degree centrality, which is the total number of nodes each node is connected with. Unfortunately, many graph theory metrics require working with adjacency matrices, which are binary matrices indicating the presence of an edge or not. To create this, we will simply apply an arbitrary threshold to our correlation matrix.

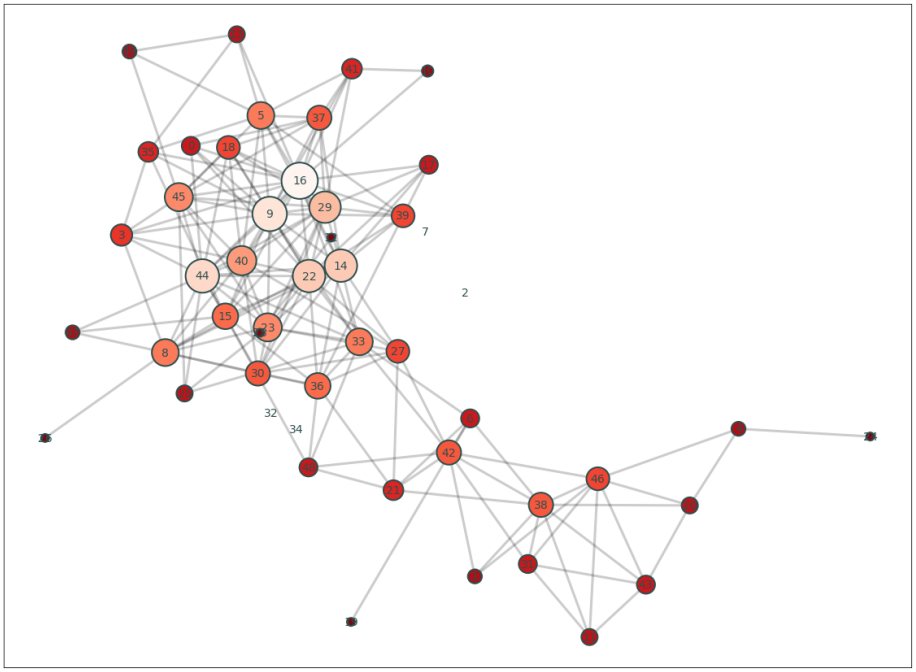
```
a = Adjacency(roi_corr, matrix_type='similarity', labels=[x for x in range(50)])
a_thresholded = a.threshold(upper=.6, binarize=True)

a_thresholded.plot()
```



Okay, now that we have a thresholded binary matrix, let's cast our data into a networkx object and calculate the degree centrality of each ROI and make a quick plot of the graph.

```
plt.figure(figsize=(20,15))
G = a_thresholded.to_graph()
pos = nx.kamada_kawai_layout(G)
node_and_degree = G.degree()
nx.draw_networkx_edges(G, pos, width=3, alpha=.2)
nx.draw_networkx_labels(G, pos, font_size=14, font_color='darkslategray')

nx.draw_networkx_nodes(G, pos, nodelist=list(dict(node_and_degree).keys()),
                       node_size=[x[1]*100 for x in node_and_degree],
                       node_color=list(dict(node_and_degree).values()),
                       cmap=plt.cm.Reds_r, linewidths=2, edgecolors='darkslategray',
alpha=1)
```

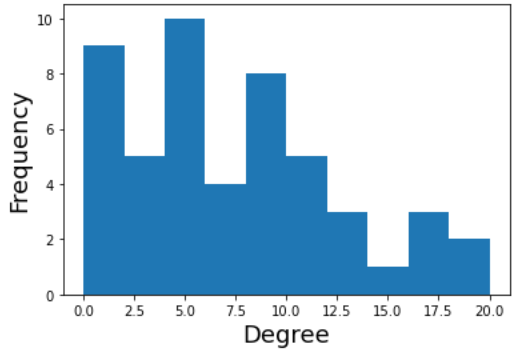
```
<matplotlib.collections.PathCollection at 0x7facb0faf810>
```



We can also plot the distribution of degree using this threshold.

```
plt.hist(dict(G.degree).values())
plt.ylabel('Frequency', fontsize=18)
plt.xlabel('Degree', fontsize=18)
```

```
Text(0.5, 0, 'Degree')
```

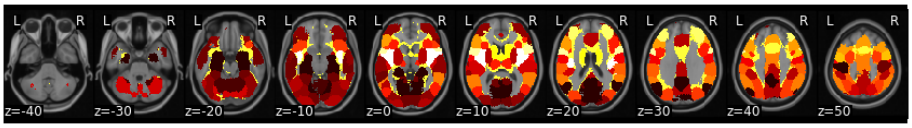


What if we wanted to map the degree of each node back onto the brain?

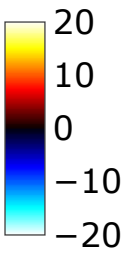
This would allow us to visualize which of the parcels had more direct pairwise connections.

To do this, we will simply scale our expanded binary mask object by the node degree. We will then combine the masks by concatenating through recasting as a brain_data object and then summing across all ROIs.

```
degree = pd.Series(dict(G.degree()))
brain_degree = roi_to_brain(degree, mask_x)
brain_degree.plot()
```




```
view_img_on_surf(brain_degree.to_nifti())
```



Left hemisphere | Inflated | view: Left

This analysis shows that the insula is one of the regions that appears to have the highest degree in this analysis. This is a fairly classic finding with the insula frequently found to be highly connected with other regions. Of course, we are only looking at one subject in a very short task (and selecting a completely arbitrary cutoff). We would need to show this survives correction after performing a group analysis.

Exercises

Let's practice what we learned through a few different exercises.

1) Let's calculate seed-based functional connectivity using a different ROI - the right motor cortex

- Calculate functional connectivity using roi=48 with the whole brain.

2) Calculate a group level analysis for this connectivity analysis

- this will require running this analysis over all subjects
- then running a one sample t-test
- then correcting for multiple tests with fdr.

3) Calculate an ICA

- run an ICA analysis for subject01 with 5 components
- plot each spatial component and its associated timecourse

4) Calculate Eigenvector Centrality for each Region

- figure out how to calculate eigenvector centrality and compute it for each region.

5) Calculate a group level analysis for this graph theoretic analysis

- this will require running this analysis over all subjects

- then running a one sample t-test
- then correcting for multiple tests with fdr.

Multivariate Prediction

Written by Luke Chang

The statistical methods we have discussed in this course so far have primarily been concerned with modeling activation in a *single voxel* and testing hypotheses in the form of "where in the brain is activation significantly greater in condition A relative condition B". As you may recall this involved using multilevel modeling with the GLM, where a voxel's time course was modeled using a first level GLM and then the contrast effect was aggregated across participants in the second level model. This procedure is often referred to as mass univariate testing and requires carefully considering how to correct for the many tests across voxels.

$$\text{voxel} = \beta \cdot \text{task model} + \beta \cdot \text{covariates} + \epsilon$$

A completely different approach to the problem is to reverse the regression equation and identify patterns of voxel activations that predict an outcome. This might be **classifying** between different conditions of a task, or **predicting** the intensity of a continuous outcome measure (e.g., emotion, pain, working memory load, etc).

$$\text{outcome} = \sum_{i}^{n} \beta_i \cdot \text{voxel}_i + \epsilon$$

Here we are learning a model of β values across the brain that when multiplied by new data will predict the intensity of a psychological state or outcome or the probability of being a specific state.

$$\text{predicted outcome} = \text{model} \cdot \text{brain data}$$

This is the general approach behind *supervised learning* algorithms. The intuition behind this approach is that brain signal might not be functionally localizable to a single region, but instead might be distributed throughout the brain. Patterns of brain activity can thus be used to *decode* psychological states.

The focus of this supervised learning approach is to accurately predict or classify the outcome, whereas the goal in classical statistics is to test hypotheses about *which* regressor explains the most independent variance of the dependent variable. These two approaches are complementary, but require thinking carefully about different issues.

In mass-univariate testing, we spent a lot of time thinking carefully about independence of errors (e.g., multi-level models) and correcting for multiple hypothesis tests. In multivariate prediction/classification or **multivoxel pattern analysis** as it is often called, we need to carefully think about **feature selection** - which voxels we will include in our model, and **cross-validation** - how well our model will *generalize* to new data. In MVPA, we typically have more features (i.e., voxels) then data points (i.e., n < p), so this requires performing feature selection or a data reduction step. The algorithms used to learn patterns come from the field of machine-learning are very good at detecting patterns, but have a tendency to *overfit* the training data.

In this tutorial, we will cover the basic steps of multivariate prediction/classification:

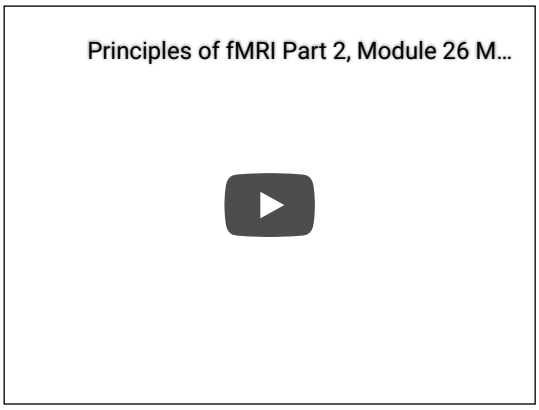
1. **Data Extraction** - what data should we use to train model?
2. **Feature Selection** - which features or voxels should we use?
3. **Cross-validation** - how do we train and test the model?
4. **Model Interpretation** - how should we interpret the model?

Why MVPA?

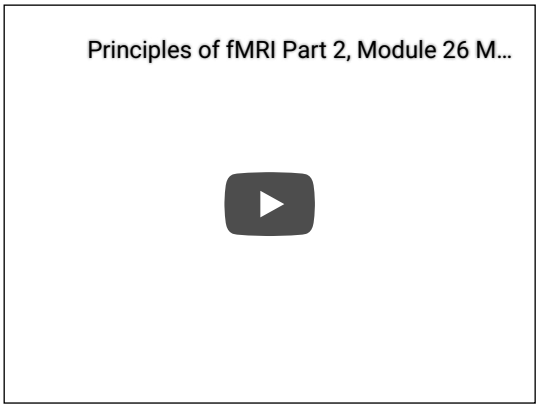
For most of our tutorials, we have tended to focus on the basics of *how* to perform a specific type of analysis, and have largely ignored questions about *why* we might be interested in specific questions. While univariate GLM analyses allow us to localize which regions might be associated with a specific psychological process, MVPA analyses, and specifically multivariate decoding, allows us to identify distributed representations throughout the brain. These might be localizable to a single region, or may be diffusely encompass many different brain systems.

Before we dive into the details of how to conduct these analyses, let' learn a little bit about the theoretical background motivating this appraoch in two videos from Tor Wager.

```
from IPython.display import YouTubeVideo

YouTubeVideo('87yKz23sPnE')
```




```
YouTubeVideo('FAyPEr7eu4M')
```



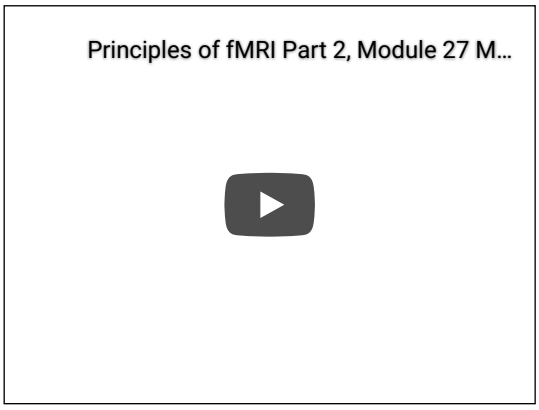
Important MVPA Concepts

Now, we are ready to dive into the details. In this tutorial, we will be using the nltools toolsbox to run these models, but also see (nilearn, brainiak, and pyMPVA) for excellent alternatives.

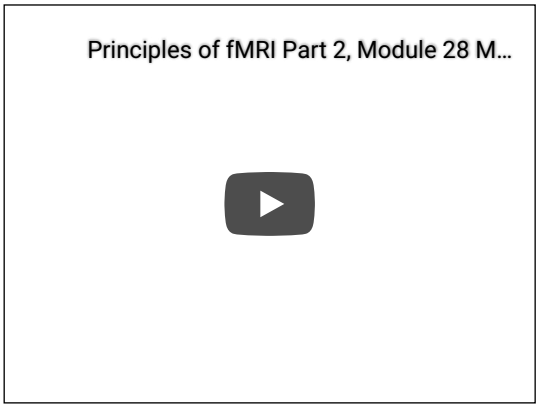
Running MVPA style analyses using multivariate regression is surprisingly easier and faster than univariate methods. All you need to do is specify the algorithm and cross-validation parameters. Currently, we have several different linear algorithms implemented from scikit-learn in the nltools package.

To make sure you understand all of the key concepts involved in the practical aspects of conducting MVPA, let's watch two short videos by Martin Lindquist before we dive into the code.

```
YouTubeVideo('dJIb5bzkQHQ')
```




```
YouTubeVideo('zKMsJyiL5Dc')
```



Data Extraction

The first step in MVPA is to decide what data you want to use to predict your outcome variable. Typically, researchers perform a temporal data reduction step, which involves estimating a standard univariate GLM using a single subject first-level model. This model will specify regressors for a single trial, or model a specific condition type over many trials. Just as in the standard univariate approach, these regressors are convolved with an HRF function. These models also usually include nuisance covariates (e.g., motion parameters, spikes, filters, linear trends, etc.). The estimated beta maps from this temporal reduction step are then used as the input data into the prediction model. Note that it is also possible to learn a *spatiotemporal* model that includes the voxels from each TR measured during a a given trial, but this is less common in practice.

First, let's load the modules we need for this analysis.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.data import Brain_Data
from nltools.mask import expand_mask
from bids import BIDSLayout, BIDSValidator
from nilearn.plotting import view_img_on_surf

data_dir = '../data/localizer'
layout = BIDSLayout(data_dir, derivatives=True)
```

Now let's load some data to train a model.

In this example, let's continue to use data from the Pinel Localizer Task that we have been using throughout all of our tutorials. For our first analysis, let's attempt to classify *Left* from *Right* motor activation. We will load a single beta image for each subject that we already estimated in earlier tutorials. We are sorting the files so that subjects are in the same order, then we are stacking all of the images together using `.append()` such that the data looks like $Subject_{1,left}, \ldots Subject_{n,left}, Subject_{1,right}, \ldots Subject_{n,right}$.

```
left_file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep',
'*','func','*_video_left*.nii.gz'))
left_file_list.sort()
left = Brain_Data(left_file_list)

right_file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep',
'*','func','*_video_right*.nii.gz'))
right_file_list.sort()
right = Brain_Data(right_file_list)

data = left.append(right)
```

Next, we need to create the labels or outcome variable to train the model. We will make a vector of ones and zeros to indicate left images and right images, respectively.

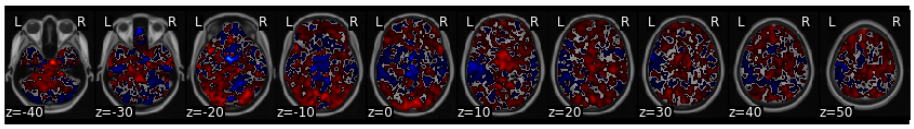
We assign this vector to the `data.Y` attribute of the Brain_Data instance.

```
Y = pd.DataFrame(np.hstack([np.ones(len(left_file_list)),
np.zeros(len(left_file_list))]))

data.Y = Y
```

okay, we are ready to go. Let's now train our first model. We will use a support vector machine (SVM) to learn a pattern that can discriminate left from right motor responses across all 9 participants.

```
svm_stats = data.predict(algorithm='svm', **{'kernel':"linear"})
```

```
overall accuracy: 1.00
```



the results of this analysis are stored in a dictionary.

- **Y**: training labels
- **yfit_all**: predicted labels
- **dist_from_hyperplane_all**: how far the prediction is from the classifier hyperplane through feature space, > 0 indicates left, while < 0 indicates right.
- **intercept**: scalar value which indicates how much to add to the prediction to get the correct class label.
- **weight_map**: multivariate brain model
- **mcr_all**: overall model accuracy in classifying training data

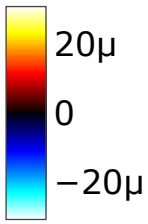
```
print(svm_stats.keys())
```

```
dict_keys(['Y', 'yfit_all', 'dist_from_hyperplane_all', 'intercept', 'weight_map',
'mcr_all'])
```

You can see that that the model can perfectly discriminate between left and right using the training data. This is great, but we definitely shouldn't get our hopes up as this model is completely being overfit to the training data. To get an unbiased estimate of the accuracy we will need to test the model on independent data.

We can also examine the model weights more thoroughly by plotting it. This shows that we see a very nice expected motor cortex representation, but notice that there are many other regions also contributing to the prediction.

```
view_img_on_surf(svm_stats['weight_map'].to_nifti())
```



Left hemisphere | Inflated | view: Left

Feature Selection

Feature selection describes the process of deciding which features to include when training the model. Here it is simply, which voxels should we use to train the model?

There are several ways to perform feature selection. Searchlights are a popular approach. I personally have a preference for using parcellation schemes.

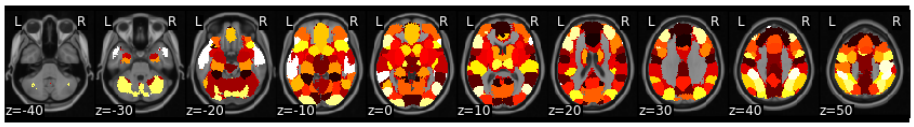
- Parcellations are orders of magnitude computationally less expensive than searchlights.
- Parcellations are easier to correct for multiple comparisons (50 vs 300k)
- Parcellations can include regions distributed throughout the brain (searchlights are only local)
- Parcellations can be integrated into a meta-model.

Here we download a single 50 parcel map from a forthcoming paper on conducting automated parcellations using neurosynth.

```
Yarkoni, T., de la Vega, A., & Chang, L.J. (In Prep).  Fully automated meta-analytic
clustering and decoding of human brain activity
```

Some of the details can be found here

```
mask = Brain_Data(os.path.join('..', 'masks', 'k50_2mm.nii.gz'))
mask_x = expand_mask(mask)

mask.plot()
```

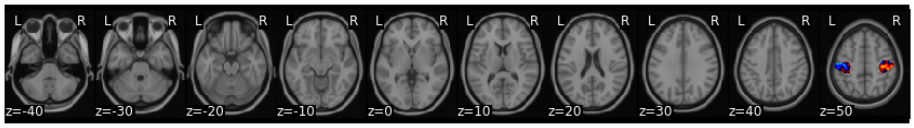


Let's combine two parcels (left-26 and right-47 motor) to make a mask and use this as a feature extract method.

This means that we will only be training voxels to discriminate between the two conditions if they are in the right or left motor cortex.

```
motor = mask_x[[26,47]].sum()

data_masked = data.apply_mask(motor)

svm_stats_masked = data_masked.predict(algorithm='svm', **{'kernel':"linear"})
```

```
overall accuracy: 1.00
```




```
svm_stats_masked['weight_map'].iplot()
```

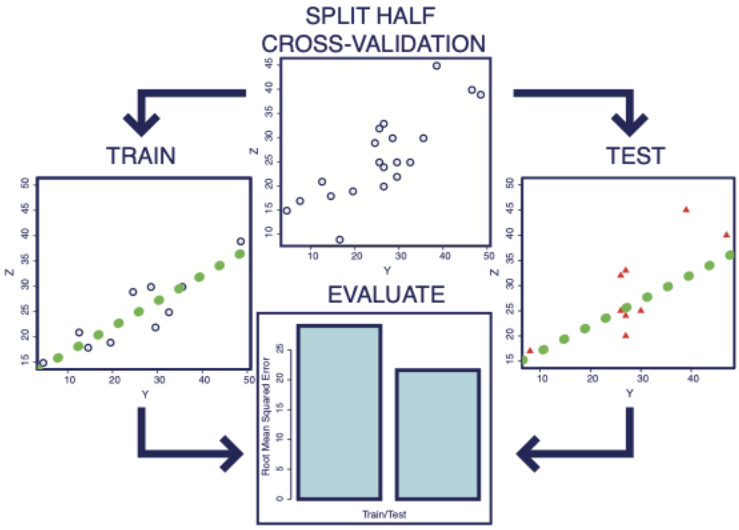
We can see that this also correctly learns that left motor cortex is positive while right cortex is negative - left vs right classification. In addition, the training accuracy is still 100%.

Cross-Validation

Clearly, our model is overfitting our training data. The next thing we need to do is to estimate how well our model will generalize to *new* data. Ideally, we would have left out some data to test after we are done training and tuning our models. This is called **holdout data** and should only be tested once when you are ready to write up your paper.

However, we don't always have the luxury of having so much extra data and also we might want to tune our model using different algorithms, features, or adjusting hyperparameters of the model.

The best way to do this, is to use **cross-validation**. The idea behind this is to subdivide the data into training and testing partitions - k-folds cross-validation is a common method - divide the data into k separate folds and use all of the data except for one fold to train the model and then test the model using the left out fold. We iterate over this process for each fold. For example, consider k=2 or split-half cross-validation.



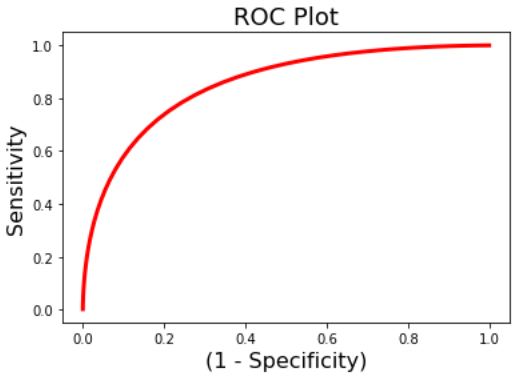
We divide the data into two partitions. We estimate the model using half of the data and test it on the other half and then evaluate how well the model performed. As you can see from this simulation, the model will almost always fit the training data better than the test data, because it is overfitting to the noise inherent to the training data, which is presumably independent across folds. More training data will lead to better estimation. This means that a k > 2 will usually result in better model estimates. When k=number of subjects, we call this *leave-one-subject-out* cross-validation.

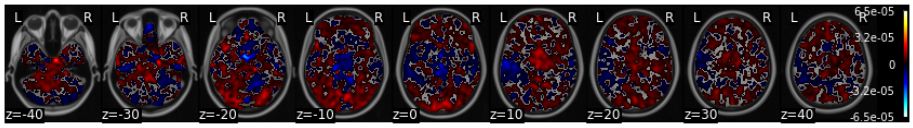
One key concept to note is that it is very important to ensure that the data is independent across folds or this will lead to a biased and usually overly optimistic generalization. This can happen if you have multiple data from the same participant. You will need to make sure that the data from the same participants are held out together. We can do this by passing a vector of group labels to make sure that data within the same group are held out together. Another approach is to make sure that the data is equally representative across folds. We can use something called stratified sampling to achieve this (see here for more details)

Let's add cross-validation to our SVM model. We will start with $k = 5$, and will pass a vector indicating subject labels as our grouping variable.

```
sub_list = [os.path.basename(x).split('_')[0] for x in right_file_list]
subject_id = pd.DataFrame(sub_list + sub_list)

svm_stats = data.predict(algorithm='svm', cv_dict={'type': 'kfolds','n_folds': 5,
'subject_id':subject_id}, **{'kernel':"linear"})
```

```
overall accuracy: 1.00
overall CV accuracy: 0.78
threshold is ignored for simple axial plots
```



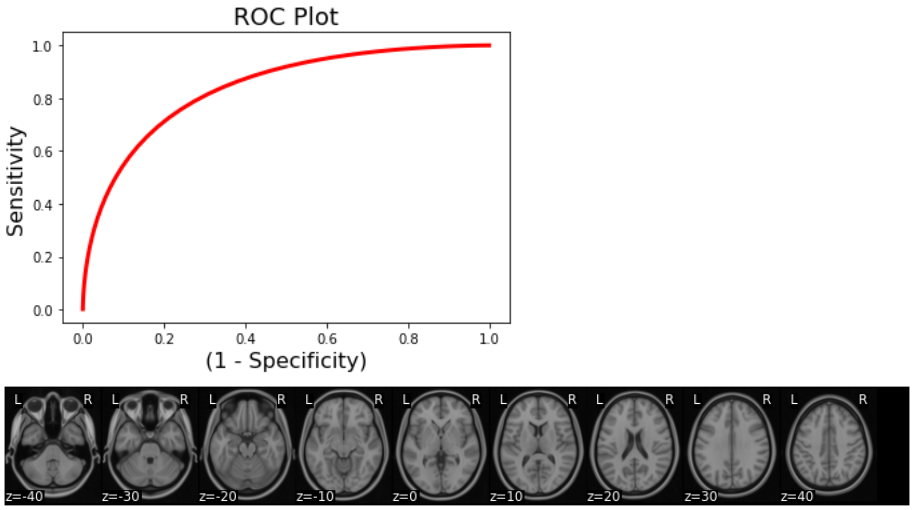


Now we see that our whole-brain model is still performing very well ~78% accuracy.

What about our masked version?

```
motor = mask_x[[26,47]].sum()
data_masked = data.apply_mask(motor)

svm_stats_masked = data_masked.predict(algorithm='svm', cv_dict={'type':
'kfolds','n_folds': 5, 'subject_id':subject_id}, **{'kernel':"linear"})
```

```
overall accuracy: 1.00
overall CV accuracy: 0.83
threshold is ignored for simple axial plots
```

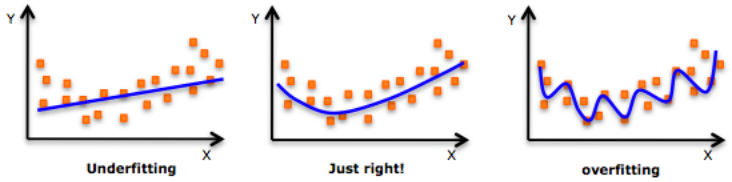


Wow, it looks like the model with feature selection actually outperforms the whole-brain model in cross-validation! 83% > 78% accuracy.

Why do you think this is the case?

Regularization

Another key concept that is used to help with feature selection is called regularization. Regularization is a method to help deal with multicollinearity and also avoid overfitting. Overfitting occurs when you have an overly complex model such as having more features(Xs) than observations(Y) (n < p). For instance, if you try to fit 10 observations with 10 features, each coefficient can be adjusted for a perfect fit but it wouldn't generalize well. In other cases, you might face the problem of feature selection. If you have numerous variables, it is time consuming to try every single combination of features in your model to see what yields the best result.



Regularization attempts to solve this problem by introducting a loss function that penalizes the model for each additional features added to the model. There are two common types of loss functions *L1* and *L2*. L1 regularization is commonly referred to as *lasso* and leads to sparse solutions, where some regressors are set to zero. L2 regularization, does not lead to a sparse solution, but instead shrinks collinear variables towards zero. Elastic Nets are a type of model that combines L1 and L2 penalizations.

Lasso regression - L1 Regularization

In short, Lasso is a feature selection method that reduces the number of features to use in a regression.
This is useful if you have a lot of variables that are correlated or you have more variables than observations.

Ridge Regression - L2 Regularization

The goal of the ridge function is to choose a penalty λ for which the coefficients are not rapidly changing and have "sensible" signs. It is especially useful when data suffers from multicollinearity, that is some of your predictor variables are highly correlated. Unlike LASSO, ridge does not produce a sparse solution, but rather shrinks variables that are highly collinear towards zero.

How do we determine the penalty value?

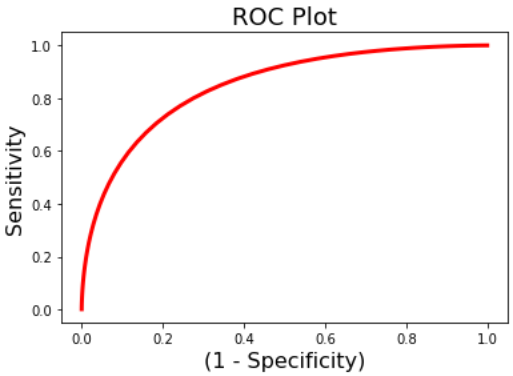
Both Lasso and Ridge regressions have a penalty hyperparameter λ. Essentially, we want to select the regularization parameter by identifying the one from a set of possible values (e.g. grid search) that results in the best fit of the model to the data. However, it is important to note that it is easy to introduce bias into this process by trying a bunch of alphas and selecting the one that works best. This can lead to optimistic evaluations of how well your model works.

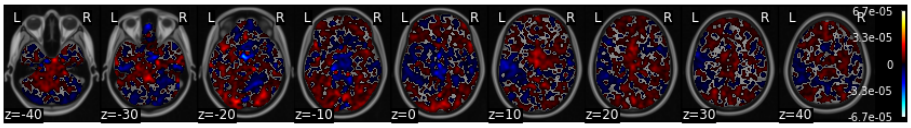
Cross-validation is an ideal method to deal with this. We can use cross-validation to select the alpha while adding minimal bias to the overall model prediction.

Here we will demonstrate using both to select an optimal value of the regularization parameter alpha of the Lasso estimator from an example provided by scikit-learn. For cross-validation, we will use a nested cross-validation as implemented by the LassoCV algorithm. Note that these examples with nested cross-validation take much longer to run.

```
ridge_stats = data.predict(algorithm='ridgeClassifier',
                           cv_dict={'type': 'kfolds','n_folds':
5,'subject_id':subject_id},
                            **{'alpha':.01})
```

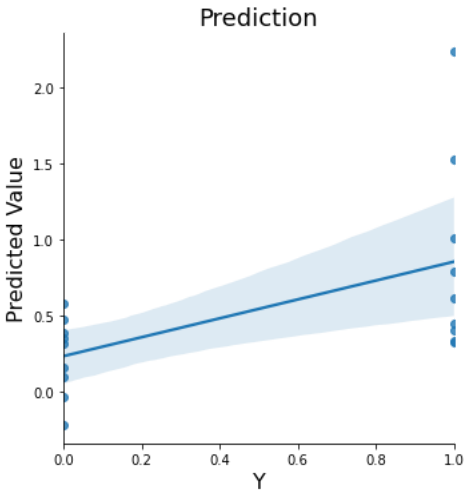
```
overall accuracy: 1.00
overall CV accuracy: 0.72
threshold is ignored for simple axial plots
```

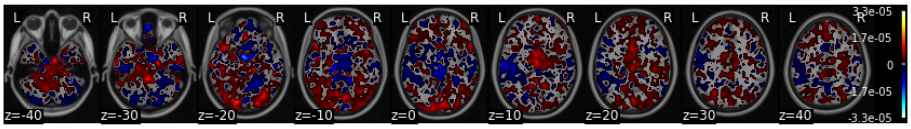





```
ridge_stats = data.predict(algorithm='ridgeCV',
    cv_dict={'type': 'kfolds','n_folds': 5, 'subject_id':subject_id})
```

```
overall Root Mean Squared Error: 0.00
overall Correlation: 1.00
overall CV Root Mean Squared Error: 0.50
overall CV Correlation: 0.56
threshold is ignored for simple axial plots
```






```
lasso_cv_stats = data.predict(algorithm='lassoCV',
    cv_dict={'type': 'kfolds','n_folds': 5, 'subject_id':subject_id})
```

Classification and Class Imbalance

One important thing to note is that when you use classification, it is important to account for class imbalances. i.e., that there might be unequal amounts of data in each group. The reason why this is a problem is that chance classification is no longer at 50% when there is a class imbalance. Suppose you were trying to classify A from B, but 80% of the data were instances of B. A classifier that always says B, would be correct 80% of the time.

There are several different ways to deal with class imbalance.

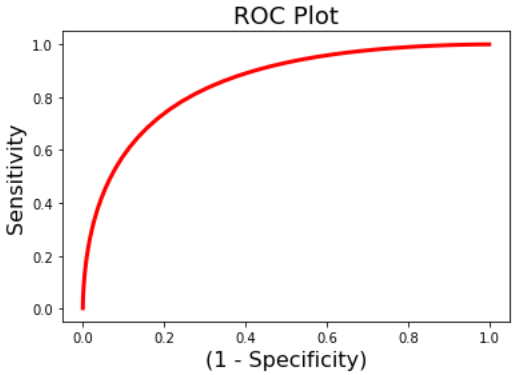
1. **Make the Class Sizes Equal** You can randomly sample data that is overrepresented to create your own balanced dataset. Advantages are that the data classes will be balanced. The disadvantage of this approach is that you are not using all of your data.
2. **Average Data** You can average all of the data within a class so that each participant only has one data point per class. Advantages are the data are balanced. Disadvantages are that you have dramatically reduced the amount of data going into training the model.
3. **Balance Class Weights** If you are using SVM, you can set `class_weight=balanced`. The general idea is to increase the penalty for misclassifying minority classes to prevent them from being "overwhelmed" by the majority class. See here for a brief overview.

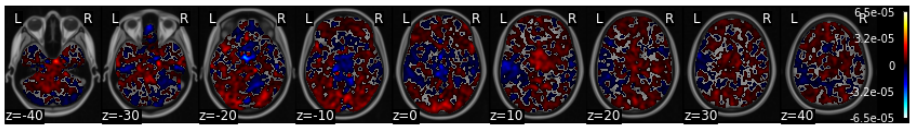
When testing your model you can also make adjustments to calculate a balanced accuracy. Scikit-learn has the `balanced_accuracy_score` method, which implements the technique outlined in this paper. It essentially defines accuracy as the average recall obtained on each class.

Let's test an example using the `'class_weight'='balanced'` approach.

```
svm_stats = data.predict(algorithm='svm', cv_dict={'type': 'kfolds','n_folds': 5,
'subject_id':subject_id}, **{'class_weight':'balanced', 'kernel':"linear"})
```

```
overall accuracy: 1.00
overall CV accuracy: 0.78
threshold is ignored for simple axial plots
```

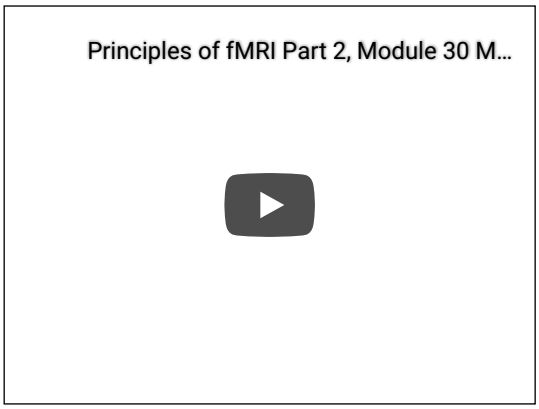




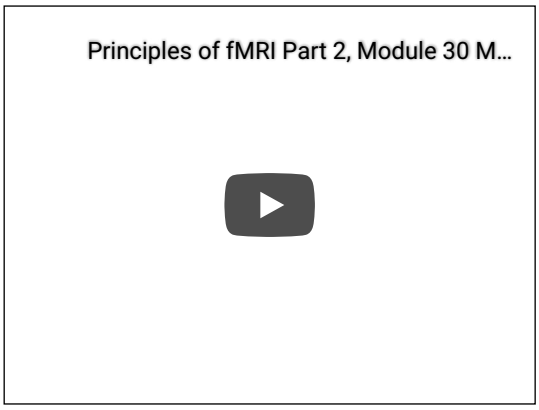
MVPA Patterns as Biomarkers

Now that we know how to train multivariate patterns, what can we do with them? There has been a lot of interest in their potential to serve as neural biomarkers of psychological states. If you would like to learn about how these can be used to better understand how we process and experience pain, watch these two videos by Tor Wager, where he summarizes some of the groundbreaking work he has been doing in this space.

```
YouTubeVideo('LV51_3jHg_c')
```




```
YouTubeVideo('3iXh0FzuAjY')
```



Additional Resources

If you are feeling like you would like to learn more about some of the details and possibilities of this approach, we encourage you to read some of the many review papers from Haynes & Rees, 2006, Naselaris et al., 2011, Haxby et al., 2014, Woo et al, 2017.

There are also many great books covering the machine-learning including the freely available the elements of statistical learning and pattern recognition and machine-learning.

Finally, Here is a helpful blog post on different algorithms and reasonable default parameters.

Exercises

Exercise 1. Vertical vs Horizontal Checkerboard Classification.

For this exercise, find a multivariate pattern that can discriminate between horizontal and vertical checkerboards in new participants with SVM and leave-one-subject out cross-validation.

Exercise 2. Generalizing Patterns.

Now, let's see how well this pattern generalizes to other conditions. See which other conditions this pattern appears to generalize too by applying the pattern to all of the participants. Does it only get *confused* for conditions involving visual information?

Representational Similarity Analysis

Written by Luke Chang

Representational Similarity Analysis (RSA) is a multivariate technique that allows one to link disparate types of data based on shared structure in their similarity (or distance) matrices. This technique was initially proposed by Nikolaus Kriegskorte in 2008. Unlike multivariate prediction/classification, RSA does not attempt to directly map brain activity onto a measure, but instead compares the similarities between brain activity and the measure using second-order isomorphisms. This sounds complicated, but is actually quite conceptually simple.

In this tutorial we will cover:

1. how to embed a stimuli in a feature space
2. how to compute similarity or distance within this feature space
3. how to map variations in position within this embedding space onto brain processes.

For a more in depth tutorial, we recommend reading this excellent tutorial written by Mark Thornton for the 2018 MIND Summer School, or watching his video walkthrough.

Let's work through a quick example to outline the specific steps involved in RSA to make this more concrete.

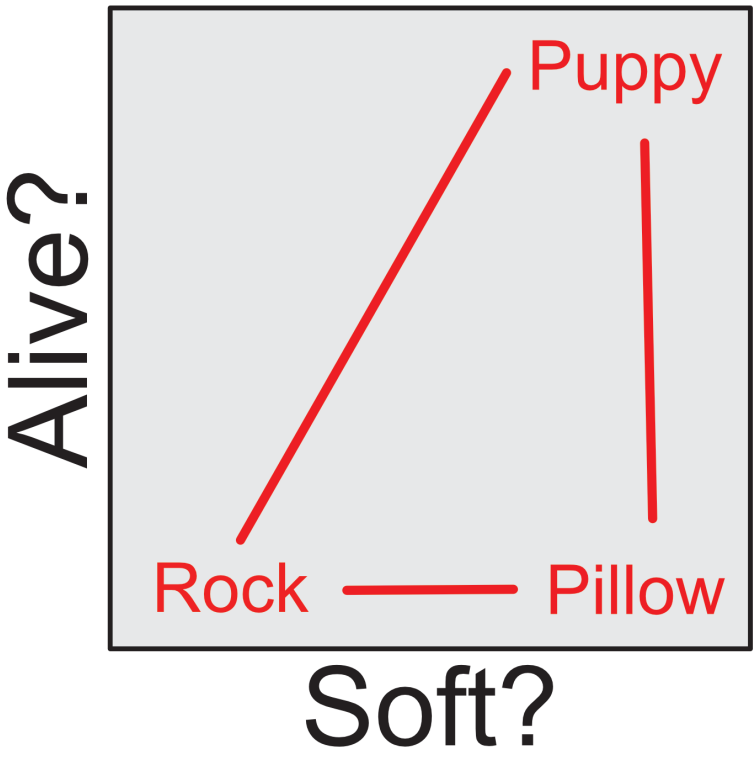
RSA Overview

Imagine that you view 96 different images in the scanner and we are interested in learning more about how the brain processes information about these stimuli. Do we categorize these images into different groups? If so, how do we do this? It might depend on what features, or aspects, of the stimuli that we consider.

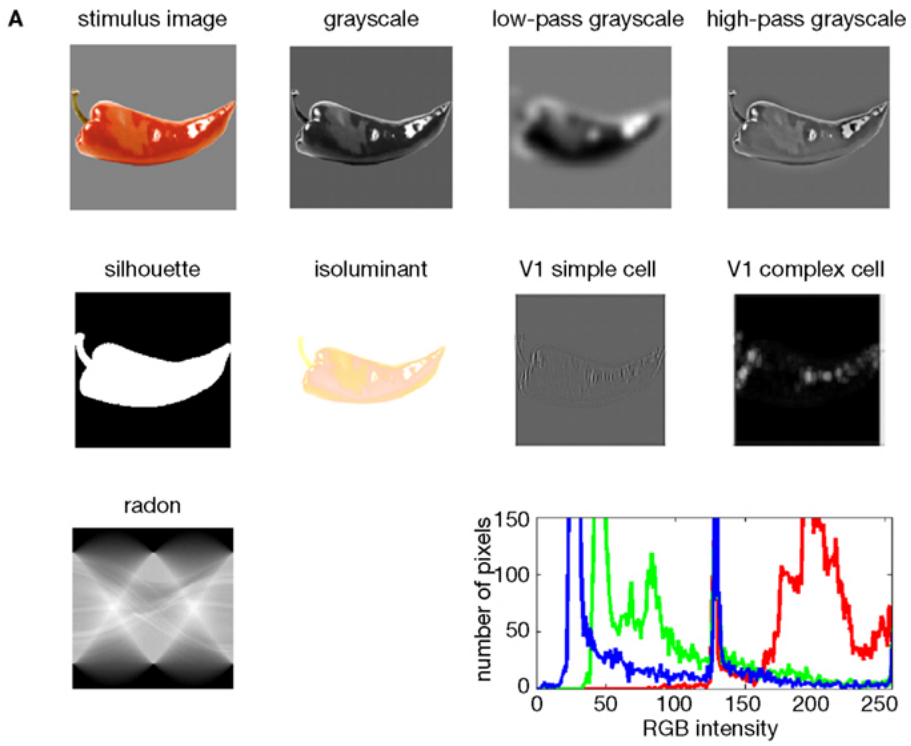
Feature embedding space

Each image can be described by a number of different variables. These variables could be more abstract, such as is it animate or inanimate? or they can be more low level, such as what color is each object?

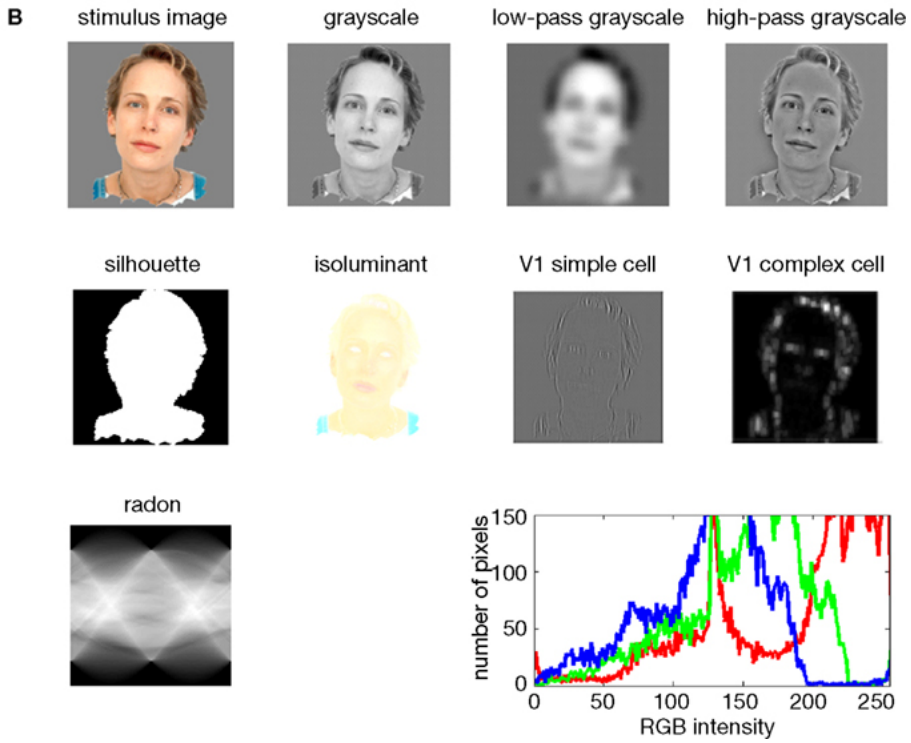
We can group together different variables into a feature space and then describe each image using this space. Think of each feature as an axis in a multidimensional space. For example, consider the two different dimensions of *animacy* and *softness*. Where would a puppy, rock, and pillow be positioned within this two dimensional embedding space?



Of course, this is a just a 2-dimensional example, this idea can be extended to n-dimensions. What if were were interested in more low-level visual features?



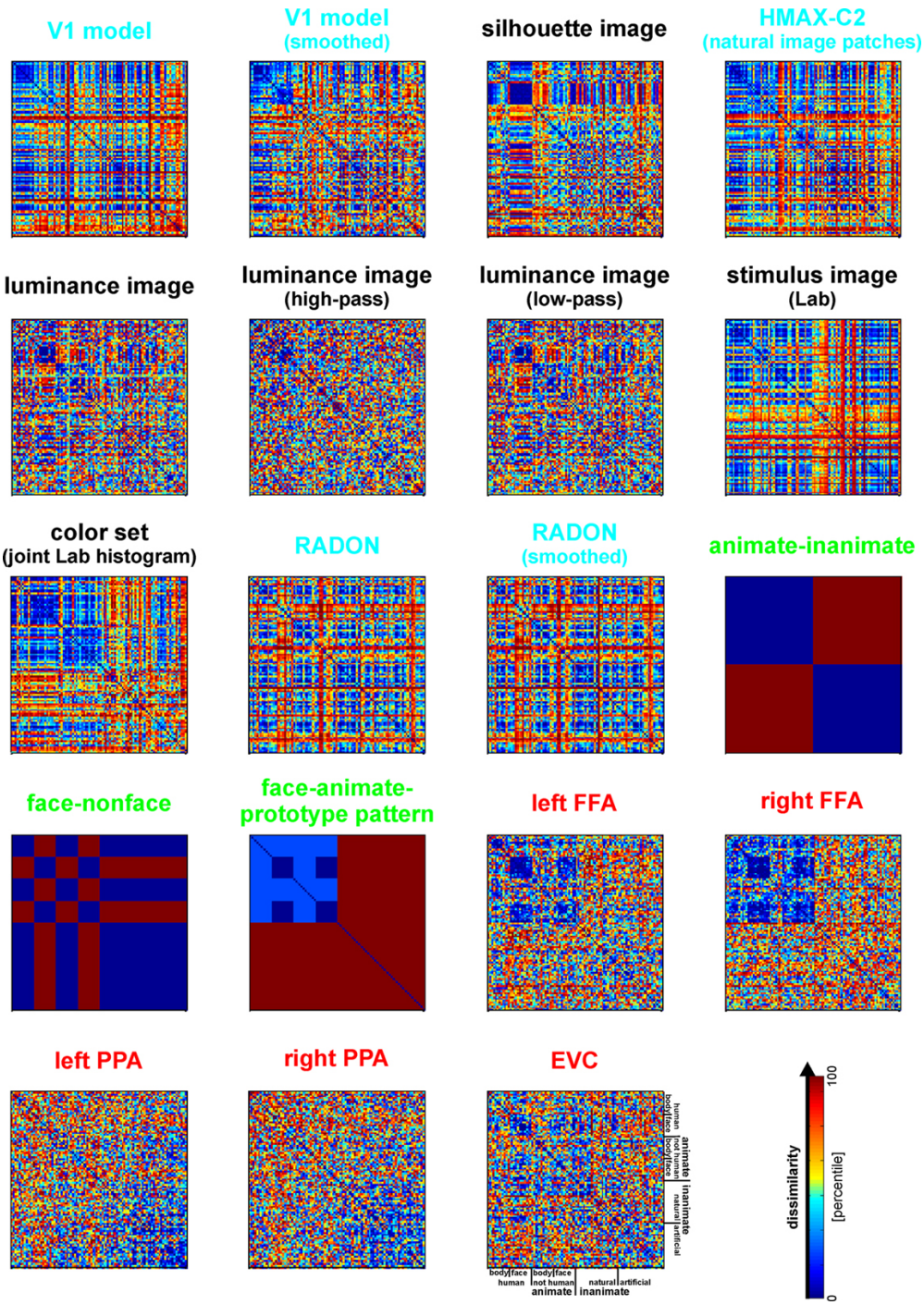
from



Kriegeskorte et al., 2008

We can compute the *representational space* of these stimuli by calculating the pairwise distance between each image in this feature space (e.g., euclidean, correlation, cosine, etc). This will yield an image x image matrix. There are many different metrics to calculate distance. For example, the absolute distance is often computed using euclidean distance, but if we don't care about the absolute magnitude in each dimension, the relative distance can be computed using correlation distance. Finally, distance is inversely proportional to similarity and these can be used relatively interchangeability (note that we will probably switch back and forth between describing similarity and distance between stimuli).

For example, if we were interested in regions that process visual information, we might extract different low-level visual features of an image (e.g., edges, luminance, salience, color, etc).

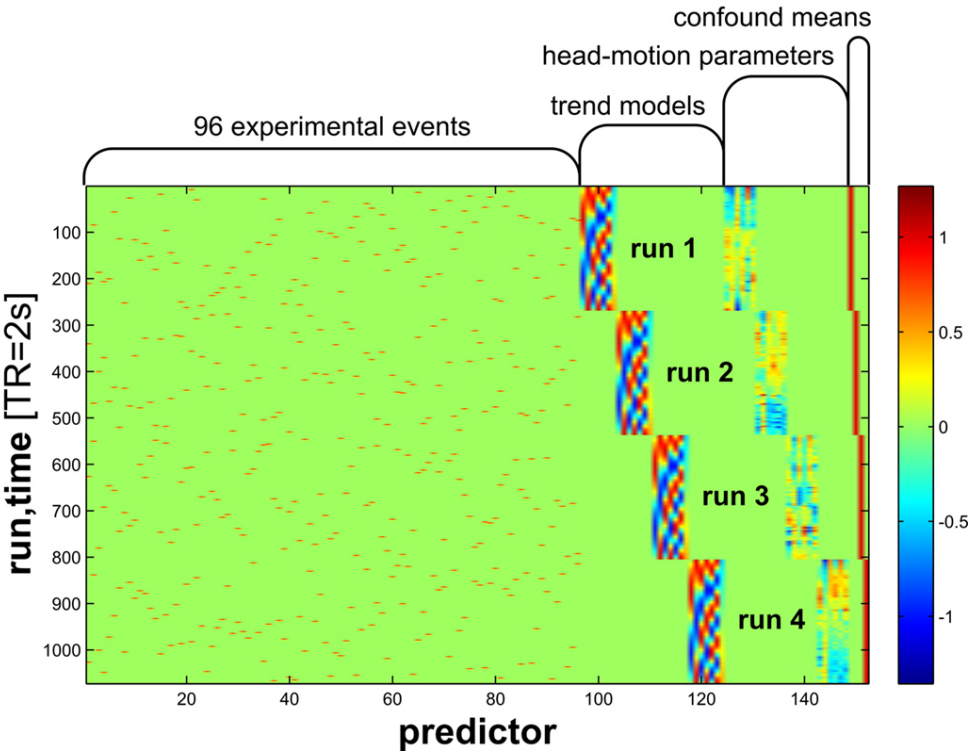


from [Kriegeskorte et al., 2008]

Brain embedding space

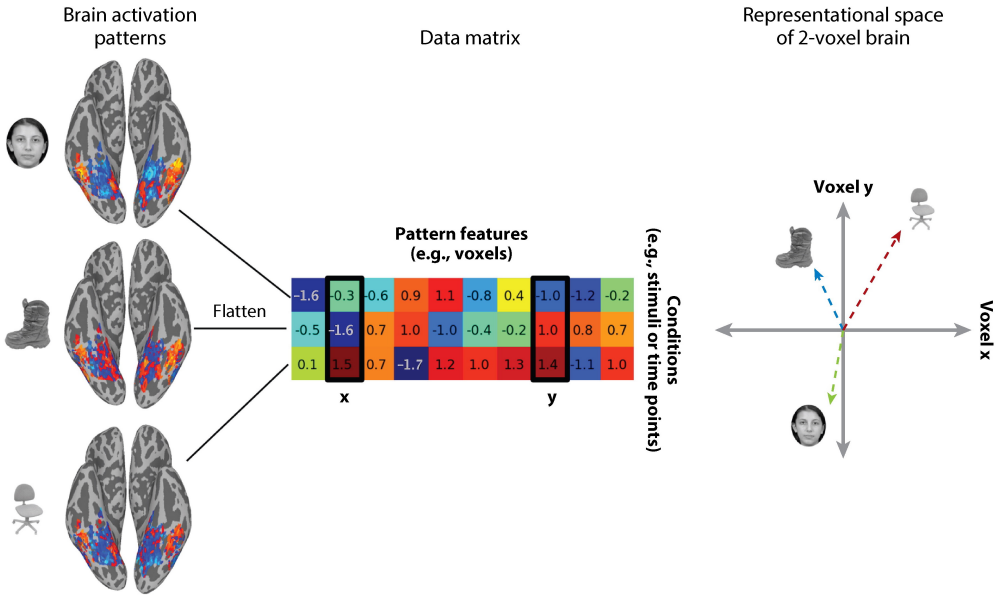
We can also embed each image in brain space. Suppose we were interested in identifying the relationship between images within the visual cortex. Here the embedding space is each voxel's activation within the visual cortex. Voxels are now the axes of the representational space.

To calculate this we need to create a brain map for every single image using a single trial model. We can use a standard first level GLM to estimate a separate beta map for each image while simultaneously removing noise from the signal by including nuisance covariates (e.g., head motion, spikes, discrete cosine transform filters, etc.)



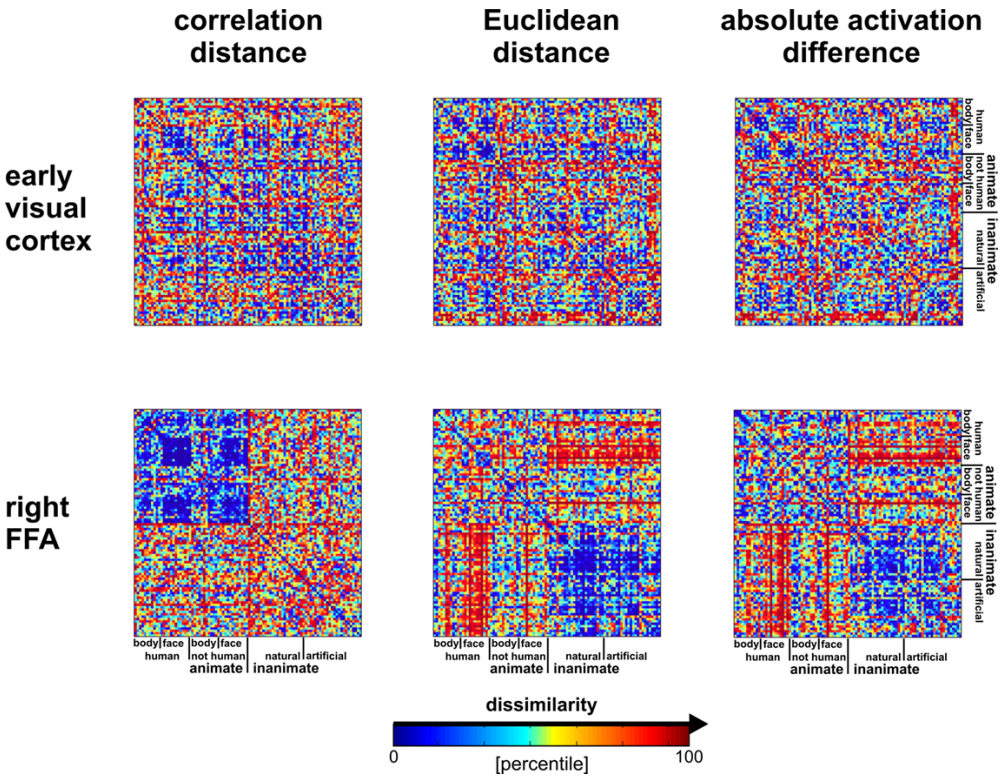
from [Kriegeskorte et al., 2008]

After we have a single beta map for each image, we can extract patterns of activity for a single ROI to yield an image by ROI voxel matrix. This allows us to calculate the representational space of how this region responds to each image by computing the pairwise similarity of the pattern of activation viewing one picture to all other pictures. In other words, each voxel within a region becomes an axis in a high dimensional space, and how the region responds to a single picture will be a point in that space. Images that have a similar response in this region will be closer in this high dimensional voxel space.



from [Haxby et al., 2014]

For fMRI, we are typically not concerned with the magnitude of activation, so we often use a relative pairwise distance metric such as correlation or cosine distance.

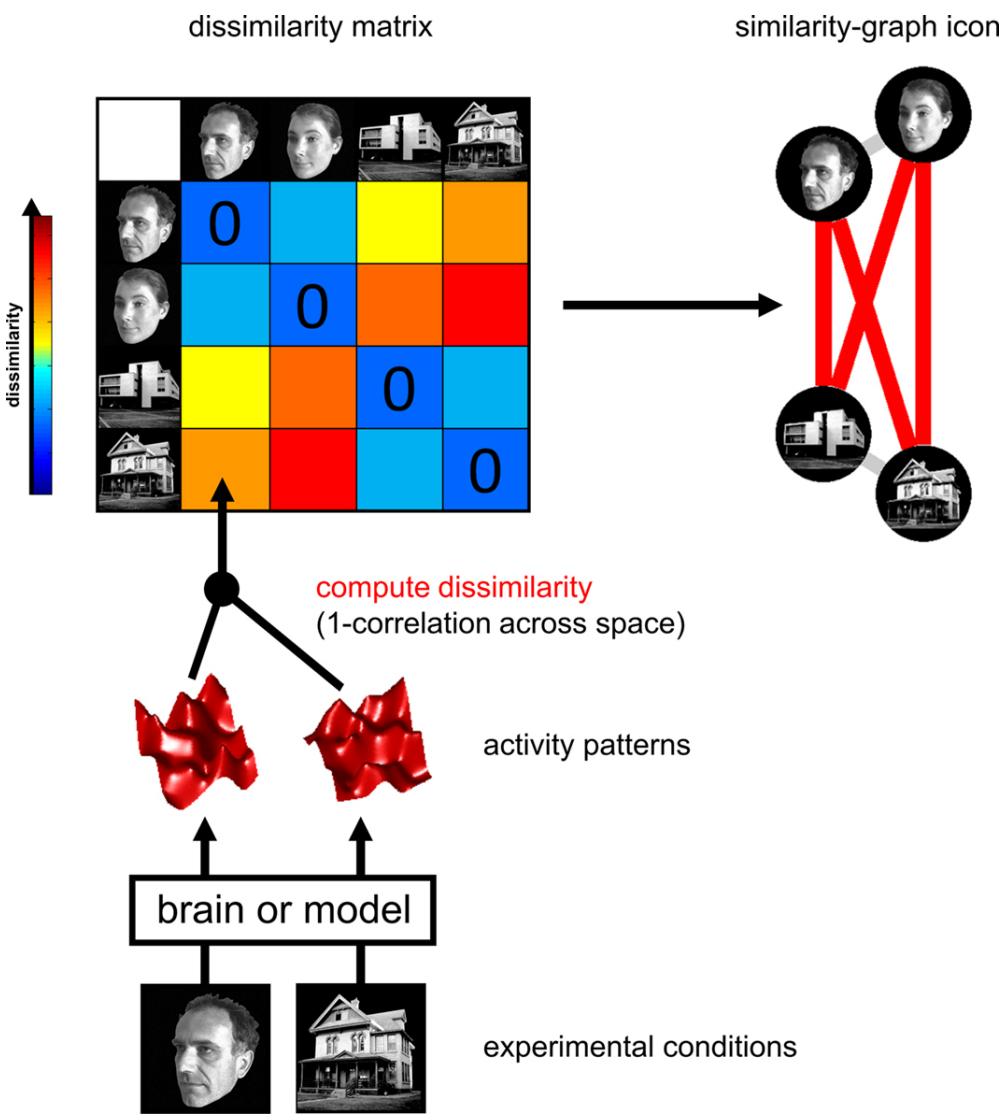


from [Kriegeskorte et al., 2008]

Mapping stimuli relationships onto brain

Now we can test different hypotheses about how the brain might be processing information about each image by mapping the representational structure of the brain space to different features spaces.

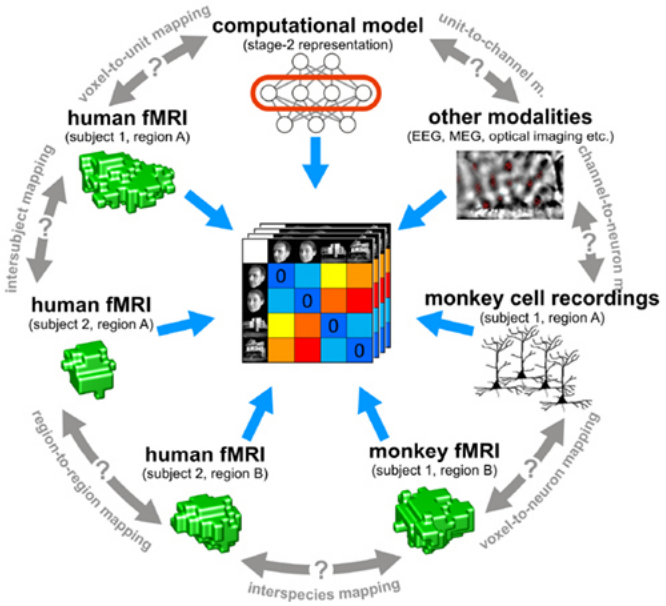
To make an inference about what each region of the brain is computing, we can evaluate if there are any regions in the brain that exhibit a similar structure as the feature representational space. Remember, these matrices are typically symmetrical (unless there is a directed relationship), so you can extract either the upper or lower triangle of these matrices. Then these triangles are flattened or vectorized, which makes it easy to evaluate the similarity between the triangle of the brain and feature matrices. Because these relationships might not necessarily be linear, it is common to use a spearman rank correlation to examine monotonic relationships between different similarity matrices.



from Kriegeskorte et al., 2008

We can make inferences across subjects, by transforming each correlation value for a given region to a continuous metric using a fisher r-to-z transformation and then computing a one-sample t-test over participants to test if the observed distribution is significantly different from zero. This is typically computed using resampling methods such as a permutation test. If inferences are over participants, then a sign test similar to a one-sample t-test is appropriate. This is a fairly standard practice when all participants viewed the same stimuli and you are interested in making inferences over participants.

There are extensions to apply this method to other types of data. For example, in Intersubject-RSA (IS-RSA), one might be interested in whether participants exhibit a particular representational structure over some feature space that maps on to inter-subject differences in brain patterns van Baar et al., 2019. In this particular, extension we cannot use the same type of permutation test to make our inferences (unless we have many different groups of participants). Instead, we might randomly shuffle one of the matrices and repeatedly re-calculate the similarity to produce an empirical distribution of similarity values. It is important to note that this type of permutation violates the exchangeability hypothesis and might yield overly optimistic p-values see Chen et al., 2017. Instead, a more conservative hypothesis testing approach is to use the mantel test, in which we only permute the rows and columns with respect to one another. Personally, I think this approach is too conservative for small sample sizes and it is still an open statistical question about how to best make inferences in this particular type of use.



from Kriegeskorte et al., 2008

One of the reasons why this technique is so powerful is that it allows the possibility of testing different computational hypotheses. Different feature representations might be associated with specific regions involved in various computations. Alternatively, different types of data can be mapped onto each other using this technique. For example, Kriegeskorte et al., 2008 have demonstrated that it is possible to map the function of IT cortex across humans and monkeys using this technique.

Now that we understand the basic steps of RSA, let's apply it to some test data. First, let's load the modules we will use for this tutorial.

```
%matplotlib inline

import os
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltools.data import Brain_Data, Adjacency
from nltools.mask import expand_mask, roi_to_brain
from nltools.stats import fdr, threshold, fisher_r_to_z, one_sample_permutation
from sklearn.metrics import pairwise_distances
from nilearn.plotting import plot_glass_brain, plot_stat_map, view_img_on_surf,
view_img
from bids import BIDSLayout, BIDSValidator

data_dir = '../data/localizer'
layout = BIDSLayout(data_dir, derivatives=True)
```

Single Subject Pattern Similarity

Recall that in the Single Subject Model Lab that we ran single subject models for 10 different regressors for the Pinel Localizer task. In this tutorial, we will use our results to learn how to conduct RSA style analyses.

First, let's get a list of all of the subject IDs and load the beta values from each condition for a single subject into a `Brain_Data` object.

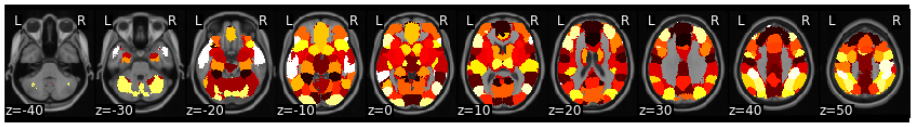
```
sub = 'S01'

file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep', f'sub-{sub}'
,'func','*denoised*.nii.gz'))
file_list = [x for x in file_list if 'betas' not in x]
file_list.sort()
conditions = [os.path.basename(x).split(f'sub-{sub}_')[1].split('_denoised')[0] for x
in file_list]
beta = Brain_Data(file_list)
```

Next we will compute the pattern similarity across each beta image. We could do this for the whole brain, but it probably makes more sense to look within a single region. Many papers use a searchlight approach in which they examine the pattern similarity within a small sphere centered on each voxel. The advantage of this approach is

that it uses the same number of voxels across searchlights and allows one to investigate the spatial topography at a relatively fine-scale. However, this procedure is fairly computationally expensive as it needs to be computed over each voxel and just like univariate analyses, will require stringent correction for multiple tests as we learned about in tutorial 12: Thresholding Group Analyses. Personally, I prefer to use whole-brain parcellations as they provide a nice balance between spatial specificity and computational efficiency. In this tutorial, we will continue to use functional regions of interest from our 50 ROI Neurosynth parcellation. This allows us to cover the entire brain with a relatively course spatial granularity, but requires several orders of magnitude of less computations than using a voxelwise searchlight approach. This means it will run much faster and will require us to use a considerably less stringent statistical threshold to correct for all independent tests. For example, for 50 tests bonferroni correction is p < 0.001 (i.e., .05/50). If we ever wanted better spatial granularity we could use increasingly larger parcellations (e.g., 100 or 200).

Let's load our parcellation mask so that we can examine the pattern similarity across these conditions for each ROI.

```
mask = Brain_Data(os.path.join('..', 'masks', 'k50_2mm.nii.gz'))
mask_x = expand_mask(mask)

mask.plot()
```



Ok, now we will want to calculate the pattern similar within each ROI across the 10 conditions.

We will loop over each ROI and extract the pattern data across all conditions and then compute the correlation distance between each condition. This data will now be an `Adjacency` object that we discussed in the Lab 13: Connectivity. We will temporarily store this in a list.

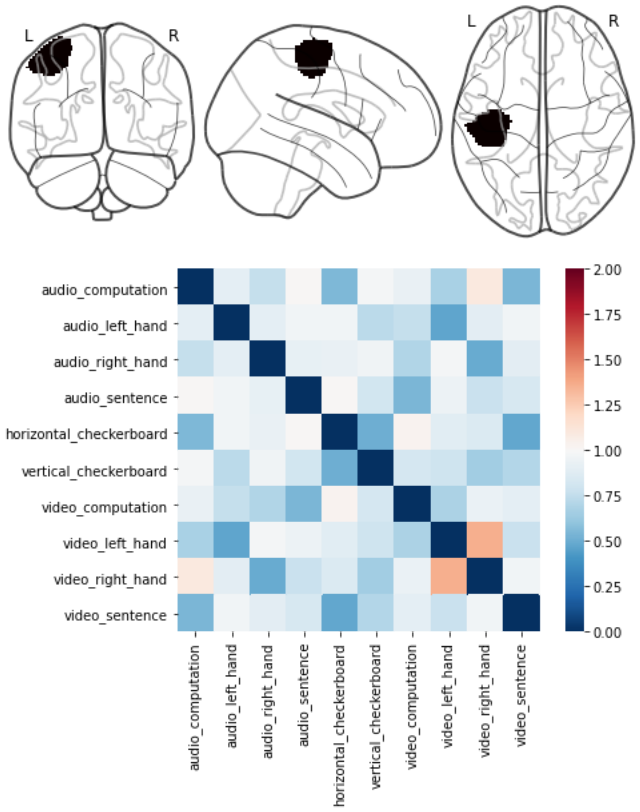
Notice that for each iteration of the loop we apply the ROI mask to our beta images and then calculate the correlation distance.

```
out = []
for m in mask_x:
    out.append(beta.apply_mask(m).distance(metric='correlation'))
```

Let's plot an example ROI and it's associated distance matrix.

Here is a left motor parcel. Notice how the distance is small between the motor left auditory and visual and motor right auditory and visual beta images?

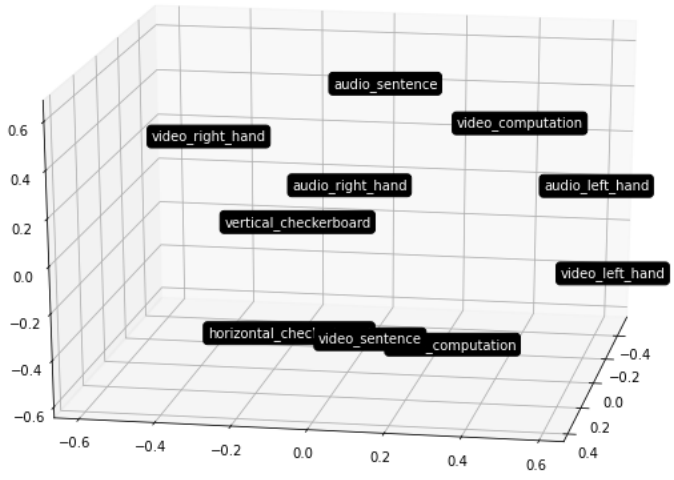
```
roi = 26
plot_glass_brain(mask_x[roi].to_nifti())
out[roi].labels = conditions
f2 = out[roi].plot(vmin=0, vmax=2, cmap='RdBu_r')
```



We can also visualize this distance matrix using multidimensional scaling with the `plot_mds()` method. This method projects the images into either a 2D or 3D plane for ease of visualization. There are many other distance based projection methods such as T-SNE or UMAP, we encourage readers to check out the excellent hypertools package that has a great implementation of all of these methods.

Notice how the motor right visual and auditory are near each other, while the motor left visual and auditory are grouped together further away?

```
f = out[roi].plot_mds(n_components=3, view=(20, 10))
```



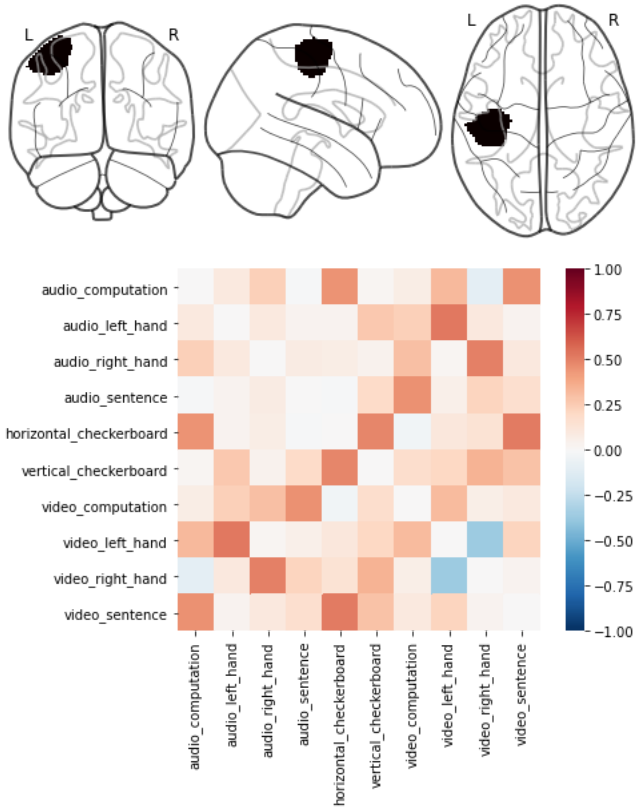
It's completely fine to continue to work with distance values, but just to make this slightly more intuitive to understand what is going on we will convert this to similarity. For correlation distance, this entails subtracting each value from 1. This will yield similarity scores in the form of pearson correlations. If you are using unbounded metrics (e.g., euclidean distance), then use the `distance_to_similarity()` Adjacency method.

We are also adding conditions as labels to the object, which make the plots easier to read.

```
out_sim = []
for m in out:
    mask_tmp = m.copy()
    mask_tmp = 1 - mask_tmp
    mask_tmp.labels = conditions
    out_sim.append(mask_tmp)
```

Let's look at the heatmap of the similarity matrix to see how more red colors indicate greater similarity between patterns within the left motor cortex across conditions.

```
roi = 26
plot_glass_brain(mask_x[roi].to_nifti())
f = out_sim[roi].plot(vmin=-1, vmax=1, cmap='RdBu_r')
```

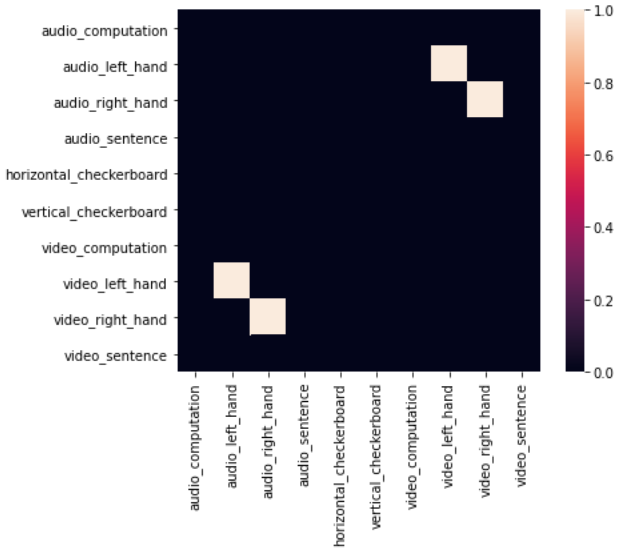


Testing a Representation Hypothesis

Ok, now that we have a sense of what the similarity of patterns look like in left motor cortex, let's create an adjacency matrix indicating a specific relationship between left hand finger tapping across the auditory and visual conditions. This type of adjacency matrix is one way in which we can test a specific hypotheses about the representational structure of the data across all images.

As you can see this only includes edges for the motor-left auditory and motor-left visual conditions.

```
motor = np.zeros((len(conditions),len(conditions)))
motor[np.diag_indices(len(conditions))] = 1
motor[1,7] = 1
motor[7,1] = 1
motor[2,8] = 1
motor[8,2] = 1
motor = Adjacency(motor, matrix_type='distance', labels=conditions)
motor.plot()
```



Now let's search over all ROIs to see if any match this representational structure of left motor-cortex using the `similarity()` method from the `Adjacency` class. This function uses spearman rank correlations by default. This is probably a good idea as we are most interested in monotonic relationships between the two similarity matrices.

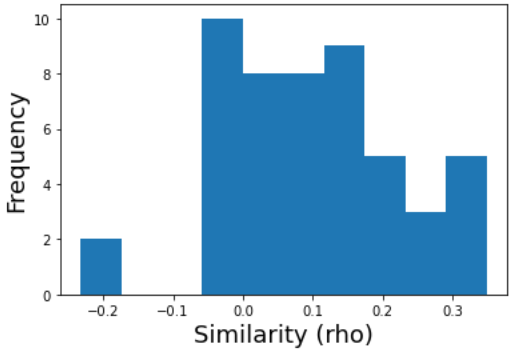
The `similarity()` method also computes permutations within columns and rows by default. To speed up the analysis, we will set the number of permutations to zero (i.e., `n_permute=0`).

```
motor_sim_r = []
for m in out_sim:
    s = m.similarity(motor, metric='spearman', n_permute=0)
    motor_sim_r.append(s['correlation'])
```

This will return a vector of similarity scores for each ROI, we can plot the distribution of these 50 ρ values.

```
plt.hist(motor_sim_r)
plt.xlabel('Similarity (rho)', fontsize=18)
plt.ylabel('Frequency', fontsize=18)
```

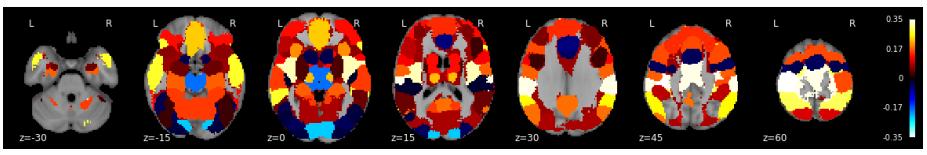
```
Text(0, 0.5, 'Frequency')
```



We can also plot these RSA values back onto the brain to see the spatial distribution.

```
rsa_motor = roi_to_brain(motor_sim_r, mask_x)

plot_stat_map(rsa_motor.to_nifti(), draw_cross=False, display_mode='z', black_bg=True,
cut_coords=np.arange(-30,70, 15))
```

```
<nilearn.plotting.displays.ZSlicer at 0x7fc04c15bd50>
```



Notice how left motor cortex is among the ROIs with the highest similarity value? Unfortunately, we can only plot the similarity values and can't threshold them yet because we didn't calculate any p-values.

We could calculate p-values using a permutation test, but this would require us to repeatedly recalculate the similarity between the two matrices and would take a long time (i.e., 5,000 correlations X 50 ROIS). Plus, the inference we want to make isn't really at the single-subject level, but across participants.

Let's now run this same analysis across all participants and run a one-sample t-test across each ROI.

RSA Group Inference

Here we calculate the RSA for each ROI for every participant. This will take a little bit of time to run (30 participants X 50 ROIs).

```
sub_list = layout.get_subjects(scope='derivatives')

all_sub_similarity = {}; all_sub_motor_rsa = {};
for sub in sub_list:
    file_list = glob.glob(os.path.join(data_dir, 'derivatives','fmriprep', f'sub-{sub}'
 ,'func','*denoised*.nii.gz'))
    file_list = [x for x in file_list if 'betas' not in x]
    file_list.sort()
    conditions = [os.path.basename(x).split(f'sub-{sub}_')[1].split('_denoised')[0] for
 x in file_list]
    beta = Brain_Data(file_list)

    sub_pattern = []; motor_sim_r = [];
    for m in mask_x:
        sub_pattern_similarity = 1 - beta.apply_mask(m).distance(metric='correlation')
        sub_pattern_similarity.labels = conditions
        s = sub_pattern_similarity.similarity(motor, metric='spearman', n_permute=0)
        sub_pattern.append(sub_pattern_similarity)
        motor_sim_r.append(s['correlation'])

    all_sub_similarity[sub] = sub_pattern
    all_sub_motor_rsa[sub] = motor_sim_r
all_sub_motor_rsa = pd.DataFrame(all_sub_motor_rsa).T
```

Now let's calculate a one sample t-test on each ROI, to see which ROI is consistently different from zero across our sample of participants. Because these are r-values, we will first perform a fisher r to z transformation. We will use a non-parametric permutation sign test to perform our null hypothesis test. This will take a minute to run as we will be calculating 5000 permutations for each of 50 ROIs (though these permutations are parallelized across cores).

```
rsa_stats = []
for i in all_sub_motor_rsa:
    rsa_stats.append(one_sample_permutation(fisher_r_to_z(all_sub_motor_rsa[i])))
```

We can plot a thresholded map using fdr correction as the threshold

```
fdr_p = fdr(np.array([x['p'] for x in rsa_stats]), q=0.05)
print(fdr_p)

rsa_motor_r = Brain_Data([x*y['mean'] for x,y in zip(mask_x, rsa_stats)]).sum()
rsa_motor_p = Brain_Data([x*y['p'] for x,y in zip(mask_x, rsa_stats)]).sum()

thresholded = threshold(rsa_motor_r, rsa_motor_p, thr=fdr_p)

plot_glass_brain(thresholded.to_nifti(), cmap='coolwarm')
```

```
-1
```

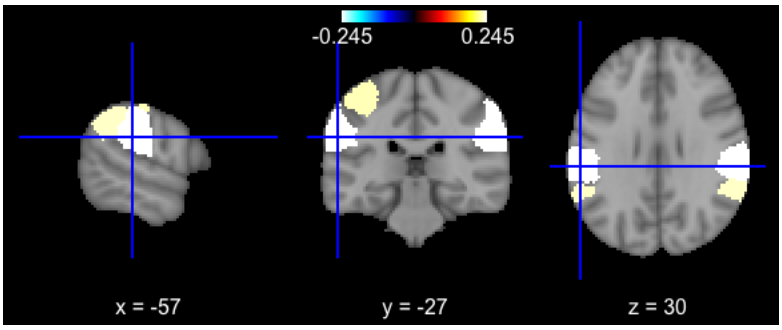
```
<nilearn.plotting.displays.OrthoProjector at 0x7fbf3929acd0>
```

Looks like nothing survives FDR. Let's try a more liberal uncorrected threshold.

```
thresholded = threshold(rsa_motor_r, rsa_motor_p, thr=0.01)

plot_glass_brain(thresholded.to_nifti(), cmap='coolwarm')
```

```
<nilearn.plotting.displays.OrthoProjector at 0x7fbf2cd13c90>
```


```
view_img(thresholded.to_nifti())
```



This looks a little better and makes it clear that the ROI that has the highest similarity with our model specifying the representational structure of motor cortex is precisely the left motor cortex. Though this only shows up using a liberal uncorrected threshold, remember that we are only using about 9 subjects for this demo.

Though this was a very simple model using a very simple dataset, hopefully this example has illustrated how straightforward it is to run RSA style analyses using any type of data. Most people use this method to examine representations of much more complicated feature spaces.

It is also possible to combine different hypotheses or models to decompose a brain representational similarity matrix. This method typically uses regression rather than correlations (see Parkinson et al., 2017)

We have recently used this technique in our own work to map similarity in brain space to individual variability in a computational model of decision-making. If you are interested in seeing an example of intersubject RSA (IS-RSA), check out the paper, and accompanying python code.

Resampling Statistics

Written by Luke Chang

Most the statistics you have learned in introductory statistics are based on parametric statistics and assume a normal distribution. However, in applied data analysis these assumptions are rarely met as we typically have small sample sizes from non-normal distributions. Though these concepts will seem a little foreign at first, I personally find them to

be more intuitive than the classical statistical approaches, which are based on theoretical distributions. Our lab relies heavily on resampling statistics and they are amenable to most types of modeling applications such as fitting abstract computational models, multivariate predictive models, and hypothesis testing.

There are 4 main types of resampling statistics:

1. **bootstrap** allows us to calculate the precision of an estimator by resampling with replacement
2. **permutation test** allows us to perform null-hypothesis testing by empirically computing the proportion of times a test statistic exceeds a permuted null distribution.
3. **jackknife** allows us to estimate the bias and standard error of an estimator by creating samples that drop one or more samples.
4. **cross-validation** provides a method to provide an unbiased estimate of the out-of-sample predictive accuracy of a model by dividing the data into separate training and test samples, where each data serves as both training and test for different models.

In this tutorial, we will focus on the bootstrap and permutation test. Jackknifing and bootstrapping are both used to calculate the variability of an estimator and often provide numerically similar results. We tend to prefer the bootstrap procedure over the Jackknife, but there are specific use cases where you will want to use the jackknife. We will not be covering the jackknife in this tutorial, but encourage the interested reader to review the wikipedia page for more information. We will also not be covering cross-validation as this is discussed in the multivariate prediction tutorial.

Bootstrap

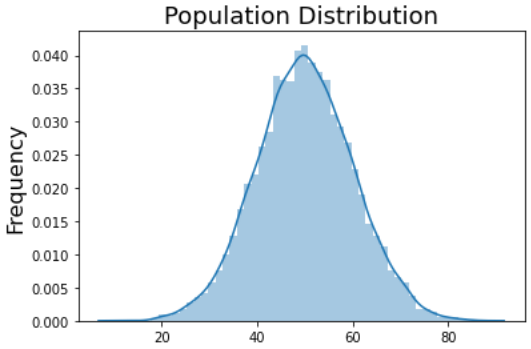
In statistics, we are typically trying to make inferences about the parameters of a population based on a limited number of randomly drawn samples. How reliable are the parameters estimated from this sample? Would we observe the same parameter if we ran the model on a different independent sample? *Bootstrapping* offers a way to empirically estimate the precision of the estimated parameter by resampling with replacement from our sample distribution and estimating the parameters with each new subsample. This allows us to capitalize on naturally varying error within our sample to create a distribution of the range of parameters we might expect to observe from other independent samples. This procedure is reasonably robust to the presence of outliers as they should rarely be randomly selected across the different subsamples. Together, the subsamples create a distribution of the parameters we might expect to encounter from independent random draws from the population and allow us to assess the precision of a sample statistic. This technique assumes that the original samples are independent and are random samples representative of the population.

Let's demonstrate how this works using a simulation.

First, let's create population data by sampling from a normal distribution. For this simulation, we will assume that there are 10,000 participants in the population that are normally distributed $\mathcal{N}(\mu = 50, \sigma = 10)$.

```
%matplotlib inline
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

mean = 50
std = 10
population_n = 10000
population = mean + np.random.randn(population_n)*std

sns.distplot(population, kde=True, label='Population')
plt.title('Population Distribution', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

print(f'Population Mean: {np.mean(population):.3}')
print(f'Population Std: {np.std(population):.3}')
```

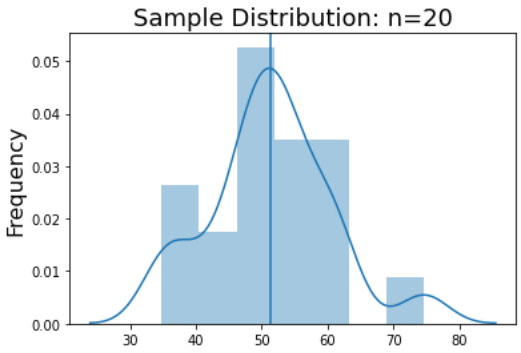
```
Population Mean: 50.0
Population Std: 10.1
```



Now, let's run a single experiment where we randomly sample 20 participants from the population. You can see that the mean and standard deviation of this distribution are fairly close to the population even though we are not full sampling the distribution.

```
sample_n = 20
sample = np.random.choice(population, size=sample_n, replace=False)

sns.distplot(sample, kde=True, label='Single Sample')
plt.axvline(x=np.mean(sample), ymin=0, ymax=1, linestyle='-')
plt.title(f'Sample Distribution: n={sample_n}', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

print(f'Sample Mean: {np.mean(sample):.3}')
print(f'Sample Std: {np.std(sample):.3}')
```

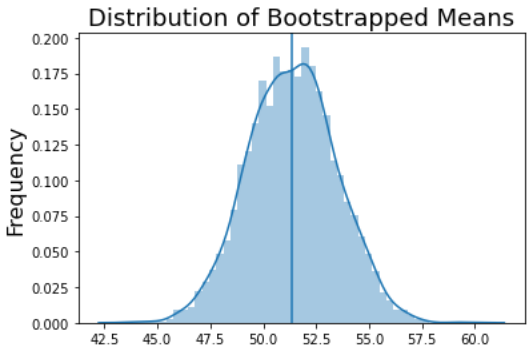
```
Sample Mean: 51.4
Sample Std: 9.18
```



Now let's estimate the mean of this sample via bootstrapping 5,000 times to estimate our certainty in this estimate from our single small sample.

```
n_bootstrap = 5000

bootstrap_means = []
for b in range(n_bootstrap):
    bootstrap_means.append(np.mean(np.random.choice(sample, size=sample_n,
replace=True)))
bootstrap_means = np.array(bootstrap_means)

sns.distplot(bootstrap_means, kde=True, label='Bootstrap')
plt.axvline(x=np.mean(bootstrap_means), ymin=0, ymax=1, linestyle='-')
plt.title('Distribution of Bootstrapped Means', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

print(f'Bootstrapped Mean: {np.mean(bootstrap_means):.3}')
print(f'Bootstrapped Std: {np.std(bootstrap_means):.3}')
```

```
Bootstrapped Mean: 51.4
Bootstrapped Std: 2.09
```

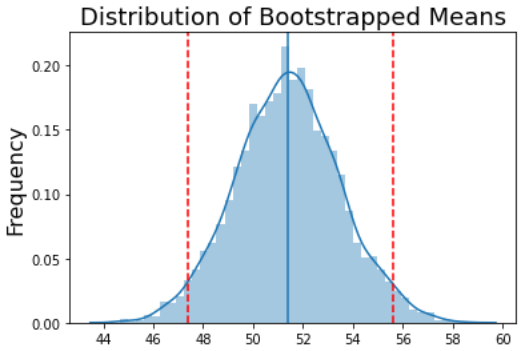


From this simulation, we can see that the mean of the bootstraps is the same as the original mean of the sample.

How confident are we in the precision of our estimated mean? In other words, if we were to look through all 5,000 of our subsamples, how many of them would be close to 50.1? We can define a confidence interval to describe our uncertainty in our estimate. For example, we can use the percentile method to demonstrate the range of the estimated parameter in 95% of our samples. To do this we compute the upper and lower quantiles of our bootstrap estimates centered at 50% (i.e., 2.5% & 97.5%).

```
n_bootstrap = 5000

bootstrap_means = []
for b in range(n_bootstrap):
    bootstrap_means.append(np.mean(np.random.choice(sample, size=sample_n,
replace=True)))
bootstrap_means = np.array(bootstrap_means)

sns.distplot(bootstrap_means, kde=True, label='Bootstrap')
plt.axvline(x=np.mean(bootstrap_means), ymin=0, ymax=1, linestyle='-')
plt.title('Distribution of Bootstrapped Means', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

lower_bound = np.percentile(bootstrap_means, 2.5)
upper_bound = np.percentile(bootstrap_means, 97.5)

plt.axvline(x=lower_bound, ymin=0, ymax=1, color='red', linestyle='--')
plt.axvline(x=upper_bound, ymin=0, ymax=1, color='red', linestyle='--')

print(f'Bootstrapped Mean: {np.mean(bootstrap_means):.3}')
print(f'95% Confidence Intervals: [{lower_bound:.3}, {upper_bound:.3}]')
```

```
Bootstrapped Mean: 51.4
95% Confidence Intervals: [47.4, 55.6]
```

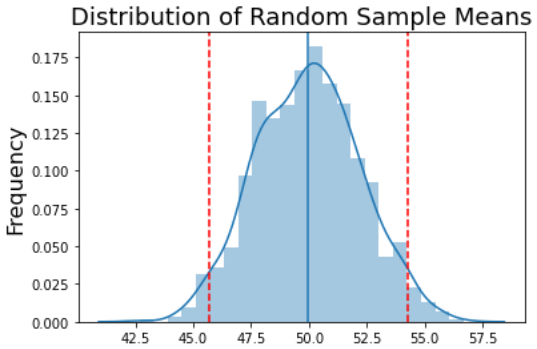


The percentile method reveals that 95% of our bootstrap samples lie between the interval [47.4, 55.6]. While the percentile method is easy to compute and intuitive to understand, it has some issues. First, if the original sample was small and not representative of the population, the confidence interval may be biased and too narrow. Second, if the bootstrapped distribution is not symmetric and is skewed, the percentile based confidence intervals will not accurately reflect the distribution. Efron (1987) proposed the bias-corrected and accelerated bootstrap, which attempts to address these issues. We will not be explaining this in detail at the moment and encourage the interested reader to review the original paper.

Now let's see how the bootstrap compares to if we had run real independent experiments. Let's simulate 1000 experiments where we randomly sample independent participants from the population and examine the distribution of the means from these independent experiments.

```
n_samples = 1000

sample_means = []
for b in range(n_samples):
    sample_means.append(np.mean(np.random.choice(population, size=sample_n,
replace=False)))
sample_means = np.array(sample_means)

sns.distplot(sample_means, kde=True, label='Random Samples')
plt.axvline(x=np.mean(sample_means), ymin=0, ymax=1, linestyle='-')
plt.title('Distribution of Random Sample Means', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

lower_bound = np.percentile(sample_means, 2.5)
upper_bound = np.percentile(sample_means, 97.5)

plt.axvline(x=lower_bound, ymin=0, ymax=1, color='red', linestyle='--')
plt.axvline(x=upper_bound, ymin=0, ymax=1, color='red', linestyle='--')

print(f'Bootstrapped Mean: {np.mean(sample_means):.3}')
print(f'95% Confidence Intervals: [{lower_bound:.3}, {upper_bound:.3}]')
```

```
Bootstrapped Mean: 50.0
95% Confidence Intervals: [45.7, 54.3]
```

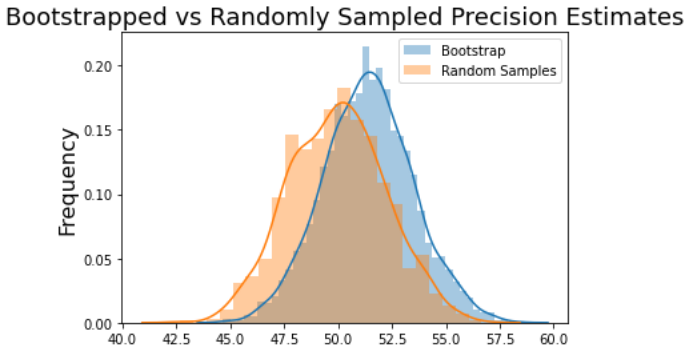


We see that the mean is closer to the population mean, but our certainty is approximately equal to what we estimated from bootstrapping a single sample.

Finally, let's compare the bootstrapped distribution of 20 samples to the 1000 random samples.

```
sns.distplot(bootstrap_means, kde=True, label='Bootstrap')
sns.distplot(sample_means, kde=True, label='Random Samples')
plt.title('Bootstrapped vs Randomly Sampled Precision Estimates', fontsize=18)
plt.ylabel('Frequency', fontsize=16)
plt.legend(['Bootstrap', 'Random Samples'])
```

```
<matplotlib.legend.Legend at 0x7fc2f92cdf10>
```



This technique is certainly not perfect, but it is very impressive how well we can estimate the precision of a population level statistic from a single small experiment using bootstrapping. Though our example focuses on estimating the mean of a population, this approach should work for many different types of estimators. Hopefully, you can see how this technique might be applied to your own work.

Permutation Test

After we have estimated a parameter for a sample, we often want to perform a hypothesis test to assess if the observed distribution is statistically different from a null distribution at a specific alpha criterion (e.g., p < 0.05). This is called null hypothesis testing, and classical statistical tests, such as the t-test, F-test, and χ^2 tests, rely on theoretical probability distributions. This can be problematic when your data are not well approximated by the theoretical distributions. Using resampling statistics, we can empirically evaluate the null hypothesis by randomly shuffling the labels and re-running the statistic. Assuming that the labels are exchangeable under the null hypothesis, then the resulting tests yield the exact significance levels, i.e., the number of times we observed our result by chance. This class of non-parametric tests are called permutation tests, and are also occasionally referred to as randomization, re-rerandomization, or exact tests. Assuming the data is exchangeable, permutation tests can provide a "p-value" for pretty much any test statistic regardless if the distribution is known. This provides a relatively straightforward statistic that is easy to compute and understand. Permutation tests can be computationally expensive and often require writing custom code. However, because they are independent they can be run in parallel using multiple CPUs or on a high performance computing system.

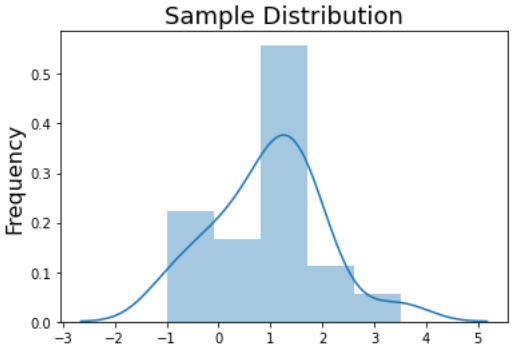
One Sample Permutation Test

Let's simulate some data to demonstrate how to run a permutation test to evaluate if the mean of the simulated sample is statistically different from zero.

We will sample 20 data points from a normal distribution, $\mathcal{N}(\mu = 1, \sigma = 1)$.

```
%matplotlib inline
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

mean = 1
std = 1
sample_n = 20
sample = mean + np.random.randn(sample_n)*std

sns.distplot(sample, kde=True, label='Population')
plt.title('Sample Distribution', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

print(f'Sample Mean: {np.mean(sample):.3}')
print(f'Sample Std: {np.std(sample):.3}')
```

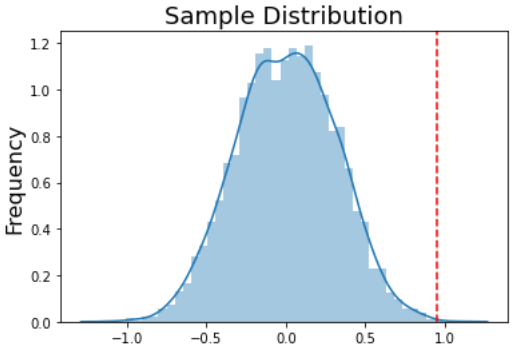
```
Sample Mean: 0.953
Sample Std: 1.04
```



In this example, the null hypothesis is that the sample does not significantly differ from zero. We can empirically evaluate this by randomly multiplying each sample by a 1 or -1 and then calculating the mean for each permutation. This will yield an empirical distribution of null means and we can evaluate the number of times the mean of our sample exceeds the mean of the null distribution.

```
n_permutations = 5000

permute_means = []
for p in range(n_permutations):
    permute_means.append(np.mean(sample * np.random.choice(np.array([1,-1]),
sample_n)))
permute_means = np.array(permute_means)

p_value = 1 - np.sum(permute_means < np.mean(sample))/len(permute_means)

sns.distplot(permute_means, kde=True, label='Population')
plt.title('Sample Distribution', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

plt.axvline(x=np.mean(sample), ymin=0, ymax=1, color='red', linestyle='--')

print(f'Sample Mean: {np.mean(sample):.3}')
print(f'Null Distribution Mean: {np.mean(permute_means):.3}')
print(f'n permutations < Sample Mean = {np.sum(permute_means < np.mean(sample))}')
print(f'p-value = {p_value:.3}')
```

```
Sample Mean: 0.953
Null Distribution Mean: 0.00544
n permutations < Sample Mean = 4998
p-value = 0.0004
```



As you can see from a null distribution that it is very rare for us to randomly observe a mean of .953. In fact, this only occurred twice in 5,000 random permutations of the data, which makes our p-value = 0.0004. The precision of our p-value is tied to the number of permutations we run. More samples will allow us to observe a higher precision of our p-value, but will also increase our computation time. We tend to use 5,000 or 10,000 permutations as defaults.

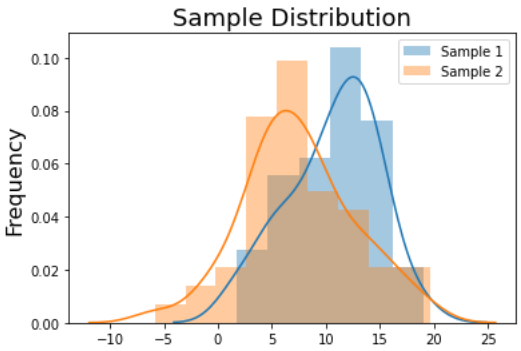
Two Sample Permutation Test

When we were computed a one-sample permutation test above, we randomly multiplied each data point by a 1 or -1 to create a null distribution. If we are interested in comparing two different groups using a permutation test, we can randomly swap group labels and can recompute the difference between the two distribution. This assumes the data are exchangeable.

Let's start by simulating two different groups. Sample 1 is randomly drawn from this normal distribution, $\mathcal{N}(\mu = 10, \sigma = 5)$, while Sample 2 is drawn from this normal distribution $\mathcal{N}(\mu = 7, \sigma = 5)$.

```
sample_n = 50

mean_1 = 10
std_1 = 5

mean_2 = 7
std_2 = 5

sample_1 = mean_1 + np.random.randn(sample_n)*std_1
sample_2 = mean_2 + np.random.randn(sample_n)*std_2

sns.distplot(sample_1, kde=True, label='Sample 1')
sns.distplot(sample_2, kde=True, label='Sample 2')
plt.title('Sample Distribution', fontsize=18)
plt.ylabel('Frequency', fontsize=16)
plt.legend(['Sample 1', 'Sample 2'])

print(f'Sample1 Mean: {np.mean(sample_1):.3}')
print(f'Sample1 Std: {np.std(sample_1):.3}')

print(f'Sample2 Mean: {np.mean(sample_2):.3}')
print(f'Sample2 Std: {np.std(sample_2):.3}')
```

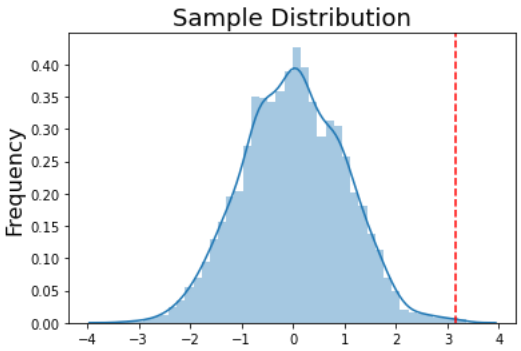
```
Sample1 Mean: 10.6
Sample1 Std: 4.15
Sample2 Mean: 7.46
Sample2 Std: 5.15
```



Ok, now to compute a permutation test to assess if the two distributions are different, we need to generate a null distribution by permuting the group labels and recalculating the mean difference between the groups.

```
data = pd.DataFrame({'Group':np.ones(len(sample_1)), 'Values':sample_1})
data = data.append(pd.DataFrame({'Group':np.ones(len(sample_2))*2,
'Values':sample_2}))

permute_diffs = []
for p in range(n_permutations):
    permutation_label = np.random.permutation(data['Group'])
    diff = np.mean(data.loc[permutation_label == 1, 'Values']) -
np.mean(data.loc[permutation_label == 2, 'Values'])
    permute_diffs.append(diff)

difference = np.mean(sample_1) - np.mean(sample_2)
p_value = 1 - np.sum(permute_diffs < difference)/len(permute_means)

sns.distplot(permute_diffs, kde=True, label='Population')
plt.title('Sample Distribution', fontsize=18)
plt.ylabel('Frequency', fontsize=16)

plt.axvline(x=difference, ymin=0, ymax=1, color='red', linestyle='--')

print(f'Difference between Sample1 & Sample2 Means: {difference:.3}')
print(f'n permutations < Sample Mean Difference = {np.sum(permute_diffs <
difference)}')
print(f'p-value = {p_value:.3}')
```

```
Difference between Sample1 & Sample2 Means: 3.16
n permutations < Sample Mean Difference = 4996
p-value = 0.0008
```



The difference we observed between the two distributions does not occur very frequently by chance. By permuting the labels and recomputing the difference, we found that the sample difference exceeded the permutated label differences 4,996/5000 times.

As long as your data are exchangeable, you can compute p-values for pretty much any type of test statistic using a permutation test. For example, to compute a p-value for a correlation between X and Y you would just shuffle one of the vectors and recompute the correlation. The p-value is the proportion of times that the correlation value exceeds the permuted correlations.

We hope that you will consider using these resampling statistics in your own work. There are many packages in python that can help you calculate bootstrap and permutation tests, but you can see that you can always write your code fairly easily if you understand the core concepts.

Spring 2019

In Spring 2019, students developed an experiment to study the neural basis of how Dartmouth undergraduates process viewing memes. Twelve Dartmouth undergraduates were scanned using functional magnetic resonance imaging (fMRI) while viewing 76 memes. After viewing each meme, participants made a decision about whether they would likely share this meme with a friend or not. After the scanning session, participants rated each meme on a number of dimensions using a Qualtrics survey (e.g., relatability, enjoyment, type of meme, etc.).

Unfortunately, we discovered that was a between run scanner artifact, which negatively impacted the results of all of the projects. Nevertheless, the questions and analytic approaches are very interesting and highlight the talent of Dartmouth undergraduates.

Self-Referencing With Memes

Sarah Eger, Taylor Walsh, Serena Zhu

Memes have become a major phenomenon in recent years, due to individuals of all ages sharingthem across social media platforms. However, no study to date has evaluated the self-referential processing that occurs in response to relatable memes. Our research undertakes this topic through the analysis of functional MRI (fMRI) scans and behavioral data gathered from ten undergraduate students at Dartmouth College. The behavioral data facilitate the conclusion that 9 of the 76 memes presented in the study directly reference Dartmouth. These data also confirm that there is a statistically significant difference in relatability between these two categories of memes. The differences in activation to stimuli belonging to each of these classes were then evaluated using a univariate analysis of the fMRI scans. This analysis showed that some activation in the dmPFC and hypothalamus survived after thresholding the contrast between Dartmouth and non-Dartmouth memes. A Multivariate Pattern Analysis (MVPA) was also implemented to determine if a Support Vector Machine (SVM) could predict which stimulus class a participant was processing, based on activation patterns in the entire brain or individual regions of interest (ROI). Both the whole-brain and 50 ROI MVPAs suggest that no particular voxel or region of the brain accounts for the classification of the two types of memes more than others. However, this pilot study was limited by a small number of subjects and some flaws in the data collection. Therefore, the findings presented in this paper should be considered only as exploratory results that merit additional inquiry through future studies of self-referential processing in relation to relatable memes, Dartmouth or otherwise.

See their `presentation`.

FFA Activation While Viewing Memes

Shivesh Shah, Hana Nazir, Huy Dang, Matthew Yuen

Human face perception is an evolutionary adaptation in humans that has become specialized because of the social nature of human life. The ability to quickly and accurately identify faces facilitates social bonding and the creation of social networks by allowing humans to gauge each other's emotions and engage in social behavior. Contemporarily, socialization has taken the form of sharing memes, or pieces of social media that convey messages that are passed on from one person to another. Understanding how viewing memes affects the brain will allow us to understand how memes provide us an avenue to socialize with each other and share commonalities. The fusiform face area (FFA) is one area that has been found to highly activate in response to faces. How this area is activated during meme-viewing has been virtually unstudied. Thus, we endeavor to uncover how the brain, specifically the FFA, reacts when viewing memes with faces and memes without faces. Our study used fMRI data when participants were exposed to memes with and without faces. We ran a univariate contrast to identify whether there was an increase in activation of the FFA in response to Face vs NoFace memes. Then, we performed a prediction analysis to see whether activation in the FFA of our participants could predict whether they perceived faces in the memes. Finally, we performed a univariate contrast of brain activation when participants viewed memes for which they perceived faces and for which they were unsure. We found that there was non-significantly more activation of the FFA when participants perceived faces than when they did not. Additionally, our model for prediction was unable to reliably predict whether participants perceived faces or not based on their brain activation. And interestingly, we found that activation was significantly greater when participants were sure they saw a face than when they were unsure. Our results provide first steps toward using memes as a way to answer broader questions on what types of materials we like, how we decide what to share, and how memes can serve as a method of strengthening social networks.

See their `presentation`.

Spring 2020

In Spring 2020, students were forced to take a remote version of the course due to the Covi19 pandemic. Unfortunately, this meant that students were unable to design and run their own original study. Instead, students developed original research questions and conducted secondary data analysis projects using data that was publicly available on OpenNeuro. One group of students worked with the naturalistic Sherlock dataset Chen et al., 2017. The other group worked with a dataset in which patient with depression and healthy controls listened to emotional music clips Lepping et al., 2016. Students were able to communicate with the authors of the study to get additional information that was not shared in the data repository to complete their analyses.

Self-Referencing With Memes

Lyrra Isanberg, Nina Kosowsky, Mia Newkirk, Kristen Soh, Sabrina Strauss

Memory is often regarded as an individualistic experience, with every person perceiving his or her world differently. To better understand the differences and similarities of human memory among individuals, we used the data collected by Chen et al. and performed several different analyses. We used contrast analyses, a representational similarity analysis (RSA), and an IS-RSA to answer several questions regarding brain activity during encoding and how that can be correlated with scene recall or semantic similarity. In our analyses, we provide answers to the following questions:

1. Do participants who successfully recall a scene have different activations during the encoding period than participants who do not recall that particular scene?
2. Are there common regions activated during encoding across subjects that are correlated with successful recall?
3. How does the temporal pattern of activity in the brain throughout encoding correlate with subject similarity, based on their scene recall?
4. How does the semantic similarity of scenes, based on their text similarity, correlate with the spatial representation of scenes during encoding?

See their `presentation`.

Music and Depression: A multivariate prediction/classification analysis and representational similarity analysis

Jada Brown, Kera Carey, Amanda Chen, Emily Chen, Mia Iqbal, Ephthalia Michael-Scwarzinger, Nathan Skinner, Bryce West

For years, music has been known to provoke a strong emotional response. Due to peoples' tendencies to self-medicate emotions with music, it has been looked at as a treatment for individuals suffering from Major Depressive Disorder (MDD) or Post-Traumatic Stress Disorder (PTSD). Emotion-provoking music has been shown to act on the reward circuitry, as well as reactivate the Anterior Hippocampus. Individuals suffering from depression and PTSD have shown damage and decreased activity in the reward system, as well as in the Hippocampus. Newer results have proved that music could be used to reactivate the Anterior Cingulate Cortex, an area with decreased activity in depressed patients. These studies have been difficult to navigate, as the stimuli, as well as cognitive reactions experienced during musical therapy, have been poorly defined. In this project, our group came up with two research questions to test by reanalzing data from (Lepping et al, 2015). (1) Can we create a model to discriminate between MDD and ND participants using the contrast between positive and negative stimuli? And (2) Can we see the differences between MDD and ND participants when processing the same audio clip? While the main study was designed to investigate neural circuitry of emotion and reward in depression, we wanted to do more with that data. Instead of just doing contrasts, and ROIs, we performed a Multivoxel Pattern Analysis (MVPA) and a Representational Similarity Analysis (RSA) using the audio files.

See their `presentation`.

Contributing

One of the wonderful aspects of both the neuroimaging and Python scientific computing communities is the strong commitment to developing and sharing knowledge and tools within the broader community. The goal of the dartbrains project is to build on this work and provide a resource for people to learn about how to analyze neuroimaging data. We try to incorporate as much open content as we can find that contributes to this goal. Please let us know if we have inadvertently omitted credit for any content generated by others. Though this project is based on a neuroimaging analysis course taught at Dartmouth College, we welcome contributions from anyone in the broader imaging community.

Getting Started

The DartBrains project is hosted on github. If you have any questions, comments, or suggestions, please open an issue.

If you notice any mistakes or have idea for new content, please either open an issue or submit a pull request for us to review.

The website is built using jupyter book, which creates a jekyll website from markdown and jupyter notebooks. Please read their materials to learn more about this neat resource.

License for this book

By Luke Chang

Supported by an NSF CAREER Award 1848370, Dartmouth Research Computing, and the Dartmouth Center for the Advancement of Learning.