NEU502B: Homework 1

The following homework assignment will require you to (1) load and visualize (f)MRI data in several different ways, (2) visualize several common confound variables, (3) create a design matrix capturing the experimental condition, and (4) use regression to model fMRI activity. Each of these problems builds on tools and ideas we've introduced in the in-class lab notebooks. We'll start by loading in some general-purpose Python modules, but you'll need to load additional modules to complete the problems (look to the lab notebooks for examples).

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from nilearn.plotting import plot_anat, plot_epi, plot_roi, plot_stat_map
from nilearn.image import mean_img, index_img
import nibabel as nib
# import matplotx
# plt.style.use(matplotx.styles.aura['dark'])
```

```
/usr/people/ye9829/.conda/envs/502b/lib/python3.9/site-packages/scipy/__init_
_.py:138: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for th
is version of SciPy (detected version 1.23.5)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is re
quired for this version of "
```

Problem 1: Visualization

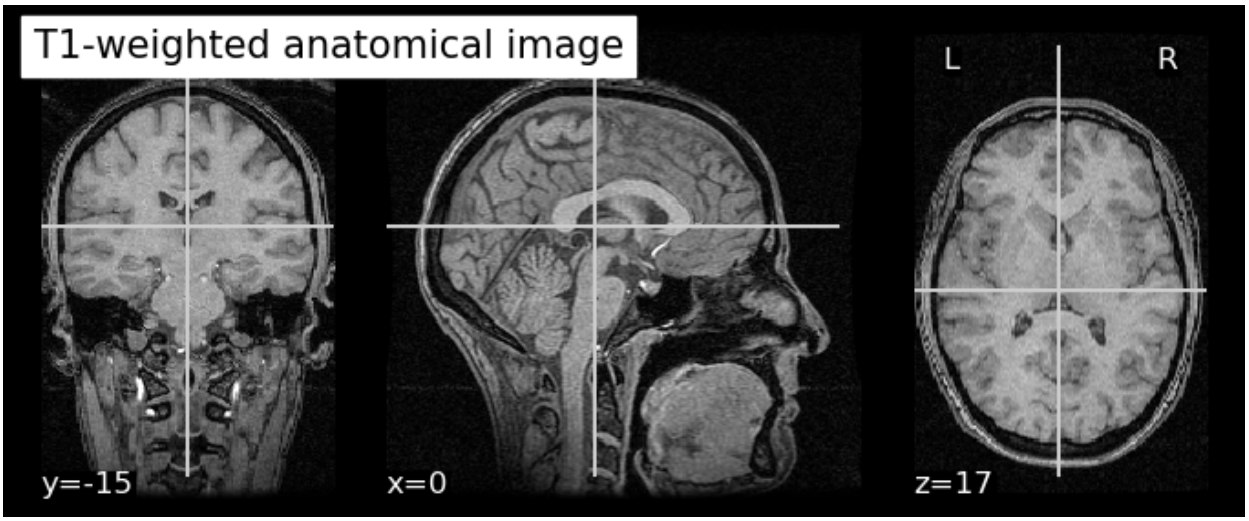
fMRI datasets are complex and noisy, so it's important to visualize your data at every stage of analysis. We'll begin by loading in the dataset from Haxby et al., 2001 using Nilearn. You'll need to change `data_dir` to a directory on your computer (or the server); if you've already downloaded this dataset in lab, you can set `data_dir` to the existing directory to save time.

In [ ]:
```
from nilearn import datasets

# Change this path to a directory on your computer!
data_dir = '/usr/people/ye9829/502b/neu502b-lab/homework'

# Load the Haxby et al., 2001 data via Nilearn
haxby_dataset = datasets.fetch_haxby(data_dir=data_dir)
```

For the first set of exercises, we'll visualize (*a*) the T1-weighted anatomical image, (*b*) the EPI image averaged across time, (*c*) an EPI volume at time point 1312, and (*d*) a mask demarcating ventral temporal (VT) cortex.

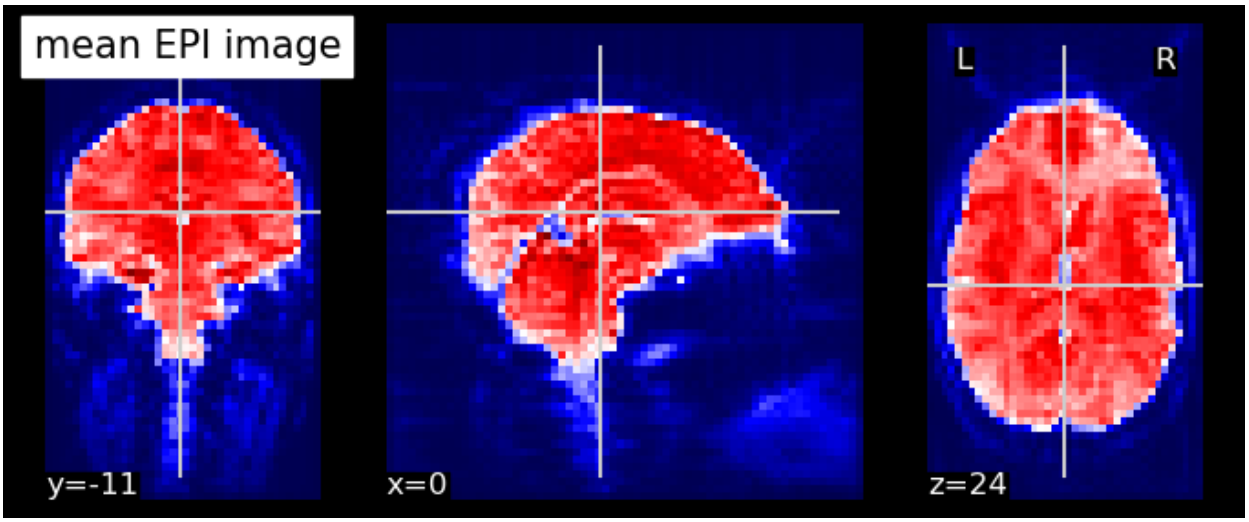
In [ ]:
```
# Visualize the T1-weighted anatomical image here:
vmin = 0
vmax = 200
plot_anat(haxby_dataset['anat'][0], title='T1-weighted anatomical image', vmin=
```

Out[ ]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7f1135c06310>
```



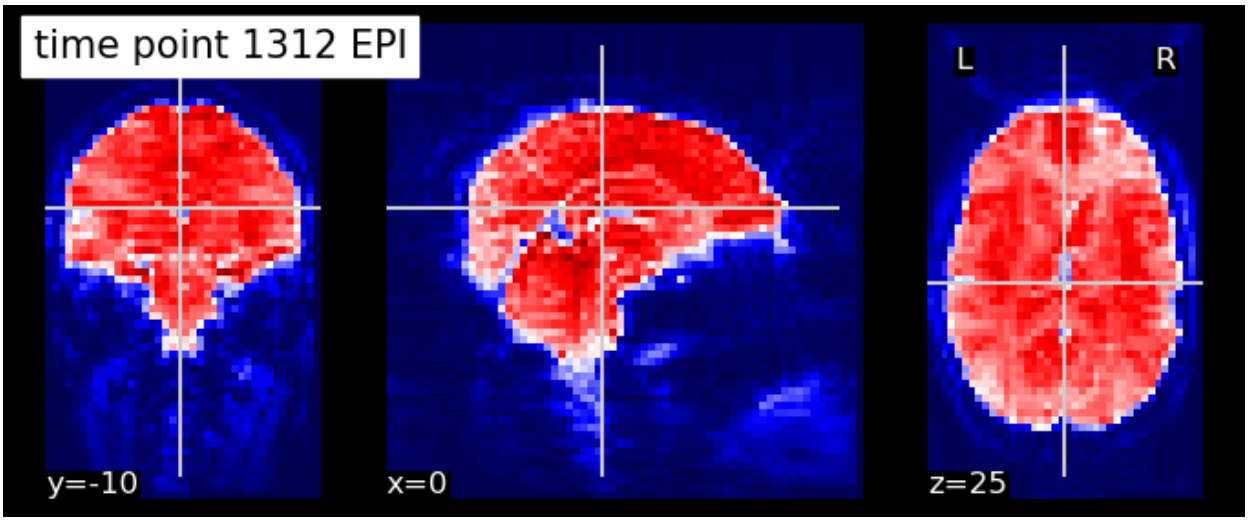
In [ ]:
```
# Visualize the mean EPI image here:
mean_epi = mean_img(haxby_dataset['func'][0])
plot_epi(mean_epi, title='mean EPI image', cmap='seismic')
```

Out[ ]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7f1138a4b160>
```



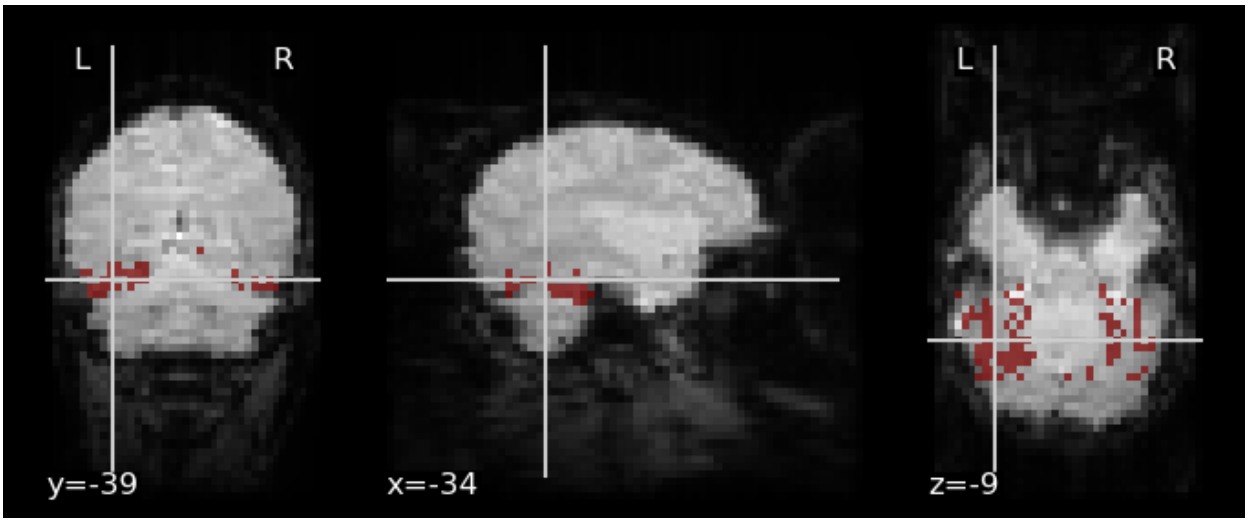
In [ ]:
```
# Visualize EPI volume (time point) 1312 here:
plot_epi(index_img(haxby_dataset['func'][0], 1312), title='time point 1312 EPI'
```

Out[ ]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7f11370ecd90>
```



In [ ]:
```
# Visualize the VT mask overlaid on the mean EPI here:
mask_vt = haxby_dataset['mask_vt'][0]
plot_roi(mask_vt, bg_img=mean_epi, cmap='seismic_r', dim=-0.1)
```

Out[ ]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7f1136f96a30>
```



Next, we'll use NiBabel to directly load in the data as NumPy arrays for manipulation in Python. First, we'll load in the VT mask as a boolean array. Inspect the shape of the functional data and mask, apply the mask to the functional data to get an array containing EPI time series for only VT cortex. Inspect the shape of the masked EPI data. Next, plot the mean time series in VT. Finally, without using Nilearn, create a "carpet plot" via Power et al., 2017 for VT data where the x-axis corresponds to time and the y-axis corresponds to voxels. Make sure to z-score each voxel's time series prior to plotting (e.g. using `zscore` from `scipy.stats`).

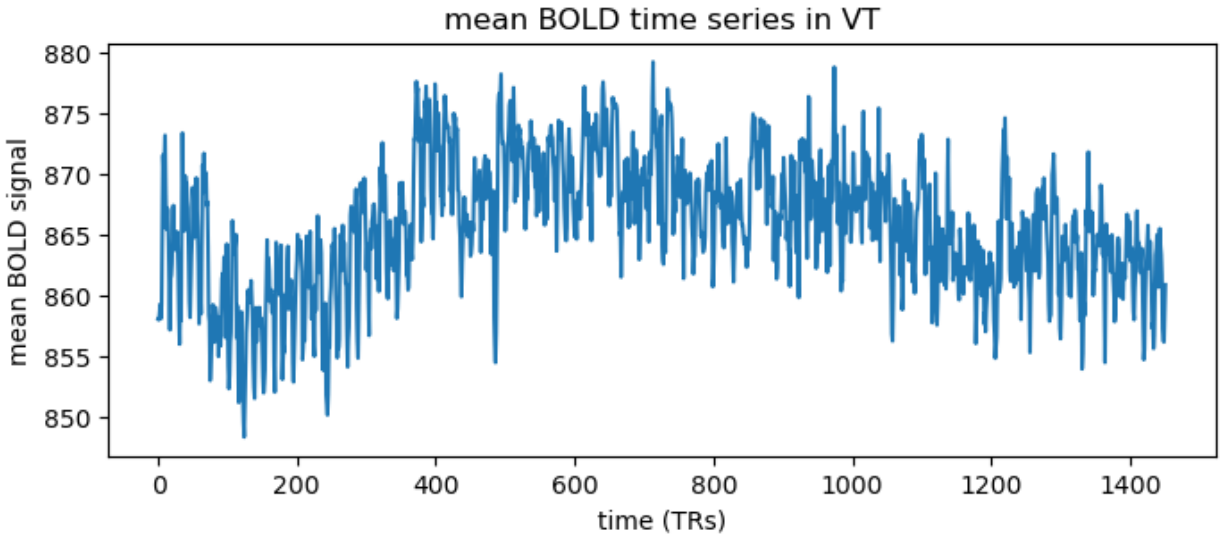
In [ ]:
```
# Use NiBabel to load functional data and VT mask:
func_data = nib.load(haxby_dataset['func'][0]).get_fdata()
vt_mask = nib.load(haxby_dataset['mask_vt'][0]).get_fdata()

# Keep the VT mask voxel indices for later:
vt_mask_idx = np.nonzero(vt_mask)
```

In [ ]:
```
# Mask functional data and inspect shapes:
func_mask = func_data[vt_mask_idx]
func_mask_full = np.ones(func_data.shape)*np.nan
func_mask_full[vt_mask_idx] = func_data[vt_mask_idx]
print(func_mask.shape)
# You may want to transpose the masked data
# so that time points are in zeroth dimension:
```

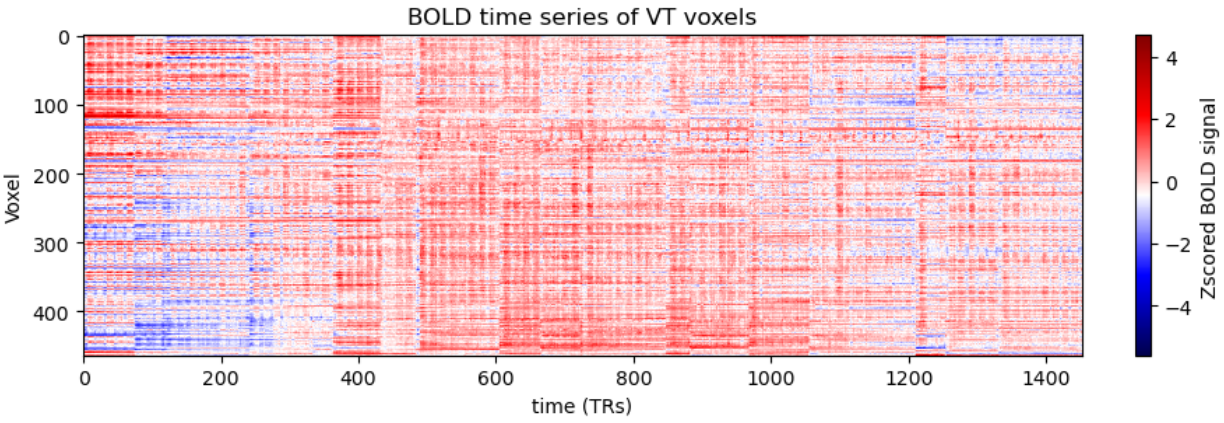
```
(464, 1452)
```

In [ ]:
```
# Plot the mean time series in VT cortex here:
fig = plt.figure(figsize=(8, 3))
ax = fig.add_subplot(title='mean BOLD time series in VT', xlabel='time (TRs)',
ax.plot(np.nanmean(func_mask, axis=(0)))
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f1136e6ad60>]
```



In [ ]:
```
from scipy.stats import zscore

# Plot carpet plot of VT data here:
zscored_func_mask = zscore(func_mask, axis=1)
fig = plt.figure(figsize=(9, 4))
ax = fig.add_subplot(title='BOLD time series of VT voxels', xlabel='time (TRs)'
img = ax.imshow(zscored_func_mask, cmap='seismic')
cbar = fig.colorbar(img, ax=ax, label='Zscored BOLD signal', fraction=0.015)
plt.tight_layout()
```



Problem 2: Confounds

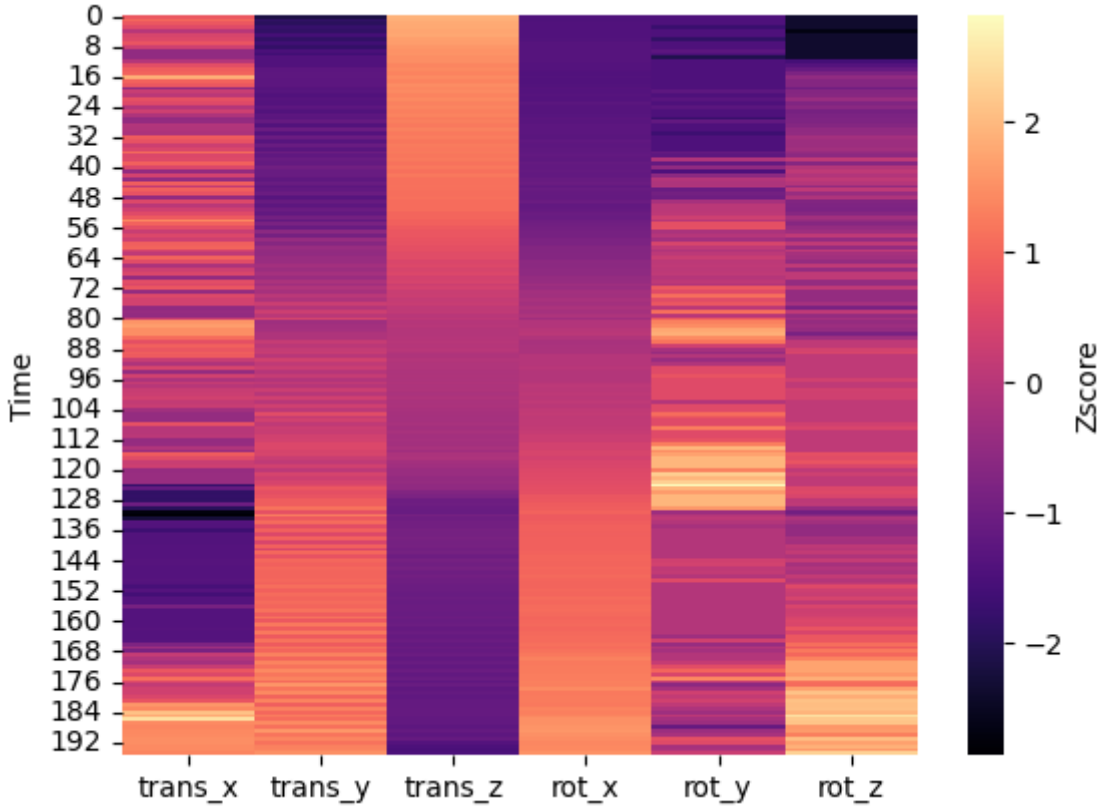
Unfortunately the publicly-available Haxby et al., 2001 dataset does not include confound variables. For this exercise, we'll take a short detour to visualize some confounds from the sample data accompanying the Princeton Handbook for Reproducible Neuroimaging. These confounds variables are created by fMRIPrep during preprocessing.

In [ ]:
```
# Load in the tabular confounds data
from pandas import read_table

confounds_f = 'sub-001_ses-01_task-faces_run-1_desc-confounds_timeseries.tsv'
df = read_table(confounds_f, sep='\t')
n_trs = len(df)
```

First, extract the six head motion parameters comprising translation (*x*-, *y*-, *z*-axes) and rotation (roll, pitch, yaw) from the counfounds table. Plot the translation and rotation time series below.

In [ ]:
```
# Extract head motion parameters and visualize:
hm_labels = ['trans_x', 'trans_y', 'trans_z', 'rot_x', 'rot_y', 'rot_z']
hm = df[hm_labels].values

ax = sns.heatmap(zscore(hm, axis=0), cmap='magma', xticklabels=hm_labels, cbar_
ax.set_ylabel('Time')
```

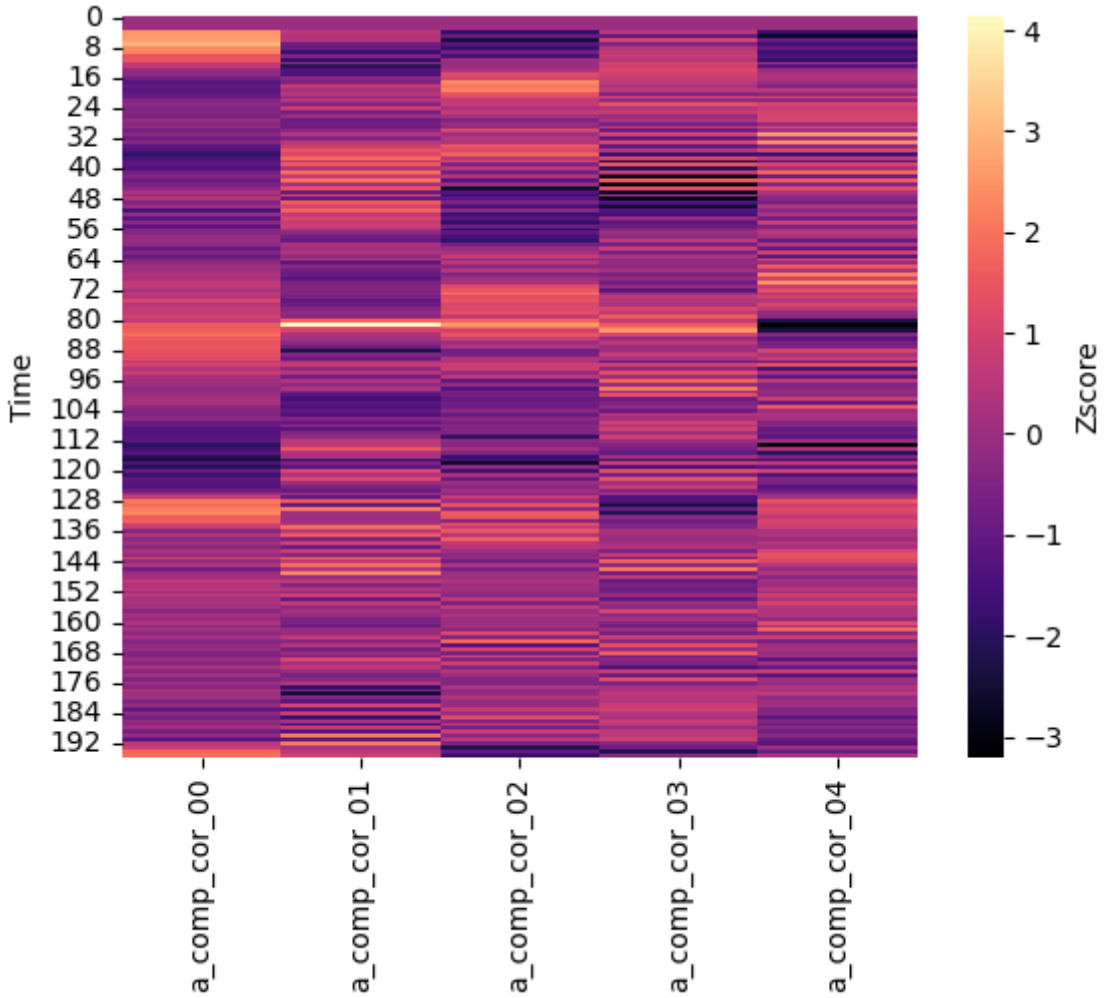
Out[ ]:
```
Text(50.722222222222214, 0.5, 'Time')
```



Next, we'll obtain the aCompCor confounds returned by fMRIPrep. These correspond to principal component (PC) time series extracted from anatomically-defined masks cerebrospinal fluid (CSF) and white matter, which may reflect physiological fluctuations and other noise sources. Extract the first 5 aCompCor time series and plot them below.

In [ ]:
```
# Extract 5 anatomical CompCor signals and plot:
acompcor_n = 5
acompcor_labels = [f'a_comp_cor_0{n}' for n in range(acompcor_n)]
acompcor = df[acompcor_labels].values

ax = sns.heatmap(zscore(acompcor, axis=0), cmap='magma', xticklabels=acompcor_l
ax.set_ylabel('Time')
```

Out[ ]:
```
Text(50.722222222222214, 0.5, 'Time')
```



fMRI data often contain slow, non-neural fluctuations over time due to thermal noise and other measurement artifacts. One way to mitigate these slow noise fluctuations is to construct detrending variables. Here, we use 4th-degree Legendre polynomials to construct a set of variables that will account for slow drifts in the signal.

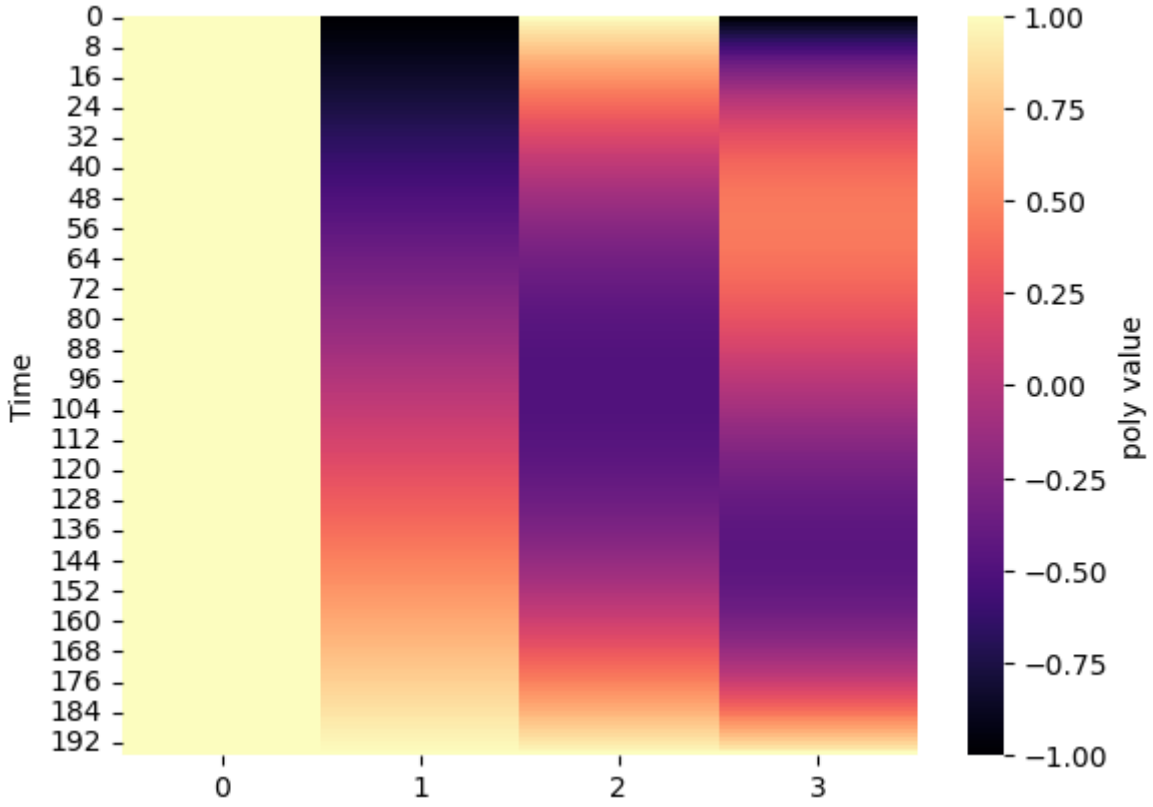
In [ ]:
```
# Construct 4th-degree Legendre polynomials
from scipy.special import eval_legendre

x_grid = np.linspace(-1, 1, n_trs)
```

```
degree = 4

polys = []
for n in np.arange(degree):
    polys.append(eval_legendre(n, x_grid))
polys = np.column_stack(polys)
```

Plot the polynomial detrending variables below on a `tr_grid` corresponding to the number of TRs.

In [ ]:
```
# Plot polynomial detrending variables on TR grid:
ax = sns.heatmap(polys, cmap='magma', cbar_kws={'label': 'poly value'})
ax.set_ylabel('Time')
```

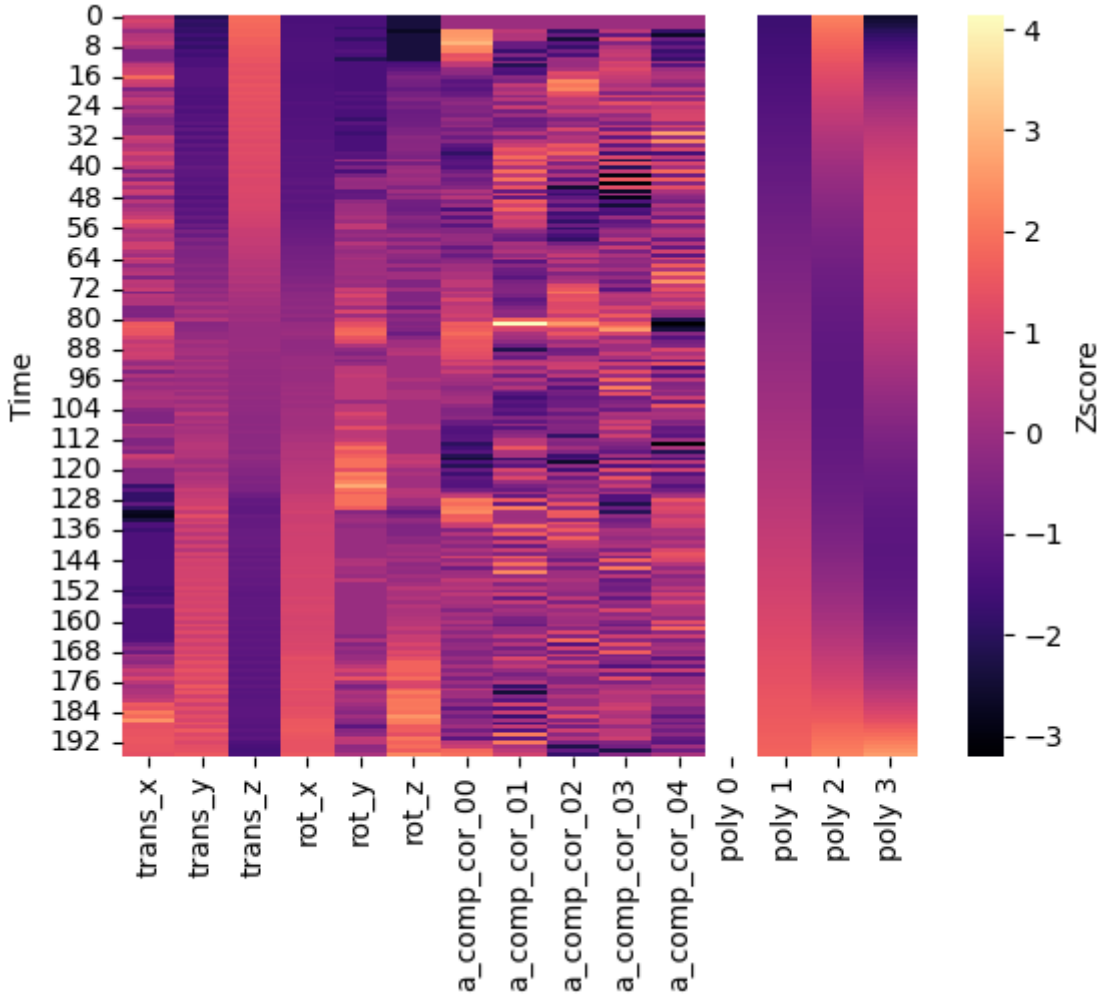
Out[ ]:
```
Text(50.722222222222214, 0.5, 'Time')
```



Finally, we'll compile all of these time series into a confound matrix. Column-stack the polynomial trends, head motion, and aCompCor variables into a single confound matrix and plot.

In [ ]:
```
# Plot confound matrix:
confounds = np.hstack((hm, acompcor, polys))
confound_labels = hm_labels + acompcor_labels + [f'poly {i}' for i in range(pol

ax = sns.heatmap(zscore(confounds, axis=0), cmap='magma', xticklabels=confound_
ax.set_ylabel('Time')
```

Out[ ]:
```
Text(50.722222222222214, 0.5, 'Time')
```



Problem 3: Design

In fMRI analysis, we usually assume that the BOLD signal is the result of a linear time-invariant (LTI) system (Boynton et al., 1996); in other words, we assume (1) the shape of the hemodynamic response is constant across time, and (2) the responses to successive stimuli superpose linearly (additively). Assumption #2 justifies using convolution in generating the predicted BOLD signal from a set of stimulus events, x, and a hemodynamic response function, h. Conceptually, what convolution does is add the entire HRF shape starting wherever there is a non-zero entry.

In the Haxby et al., 2001 experiment, participants were presented with images from 8 object categories (faces, cats, shoes, scissors, bottles, chairs, houses, and scrambled images) interspersed with periods of fixation (referred to as "rest" here). The TR in this study was 2.5 seconds. In a given run, a block of images from each of the 8 categories was presented one time. Each block was ~9 TRs long and contained multiple rapid presentations of images from a single category. A subject received 12 scanning runs. In the public dataset, these runs are concatenated into a single time series 1452 TRs in duration (although we will split it back into individual runs for certain processing steps). Below, we load in the stimulus and run labels and set some parameters for the experiment.

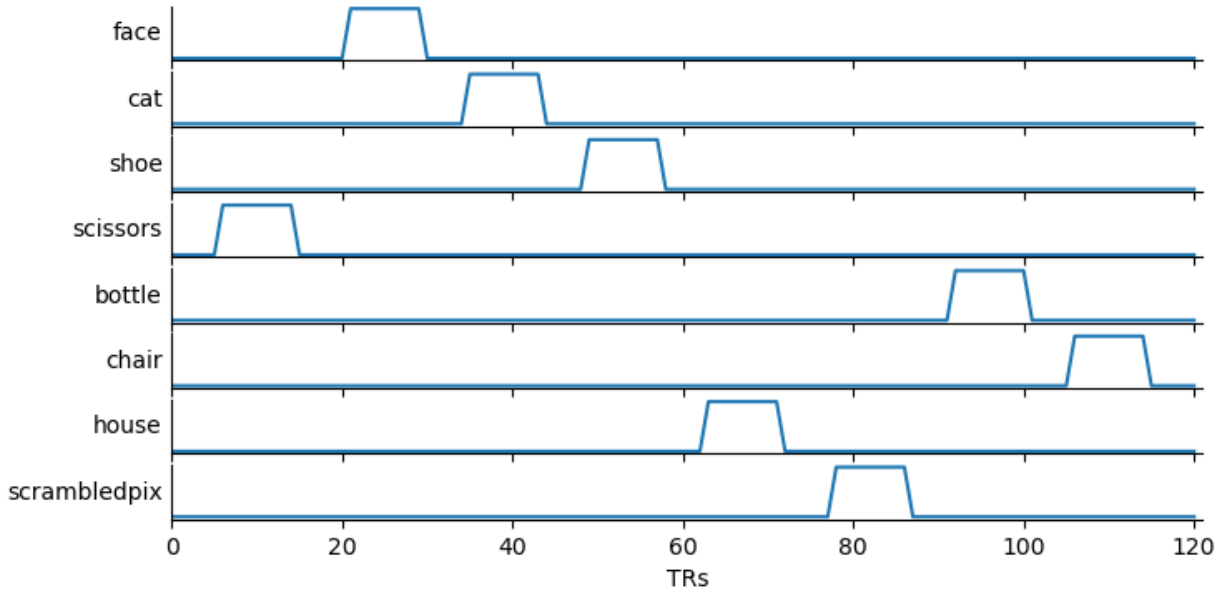
In [ ]:
```
# Load in session metadata as pandas DataFrame
session = read_table(haxby_dataset.session_target[0], sep=" ")

# Extract stimuli and run labels for this subject
stimuli, runs = session['labels'].values, session['chunks'].values

# Stimulus categories
categories = ['face', 'cat', 'shoe', 'scissors',
              'bottle', 'chair', 'house', 'scrambledpix']
n_categories = len(categories)

# Set some basic experimental parameters
tr = 2.5
n_trs = 1452
n_runs = 12
run_trs = 121
```

Next, we'll construct one boxcar time series for each of the 8 stimulus categories in the first run. Note that in this experiment each run only contains one block of stimuli per category.

In [ ]:
```
# Construct boxcar time series for the first run
run_id = 0
regressors = []
for category in categories:
    regressor = np.zeros(run_trs)
    regressor[np.where(stimuli[runs == 0] == category)] = 1
    regressors.append(regressor)
regressors = np.column_stack(regressors)
```

In the following cell, we plot the boxcar time series for each of the 8 stimulus categories. Note that we do not explicitly model the 'rest' fixation period.

In [ ]:
```
# Plot boxcar time series for the first run
fig, axs = plt.subplots(n_categories, 1, figsize=(8, 4),
                        sharex=True, sharey=True)
for i, (category, regressor) in enumerate(zip(categories, regressors.T)):
    axs[i].plot(regressor)
    axs[i].set(yticks=[])
    axs[i].set_ylabel(category, rotation=0, ha='right', va='center')
plt.xlabel('TRs')
plt.xlim(0, run_trs)
sns.despine()
```



Now, we'll load in a canonical hemodynamic response function (HRF) sampled at our 2.5-second TR.

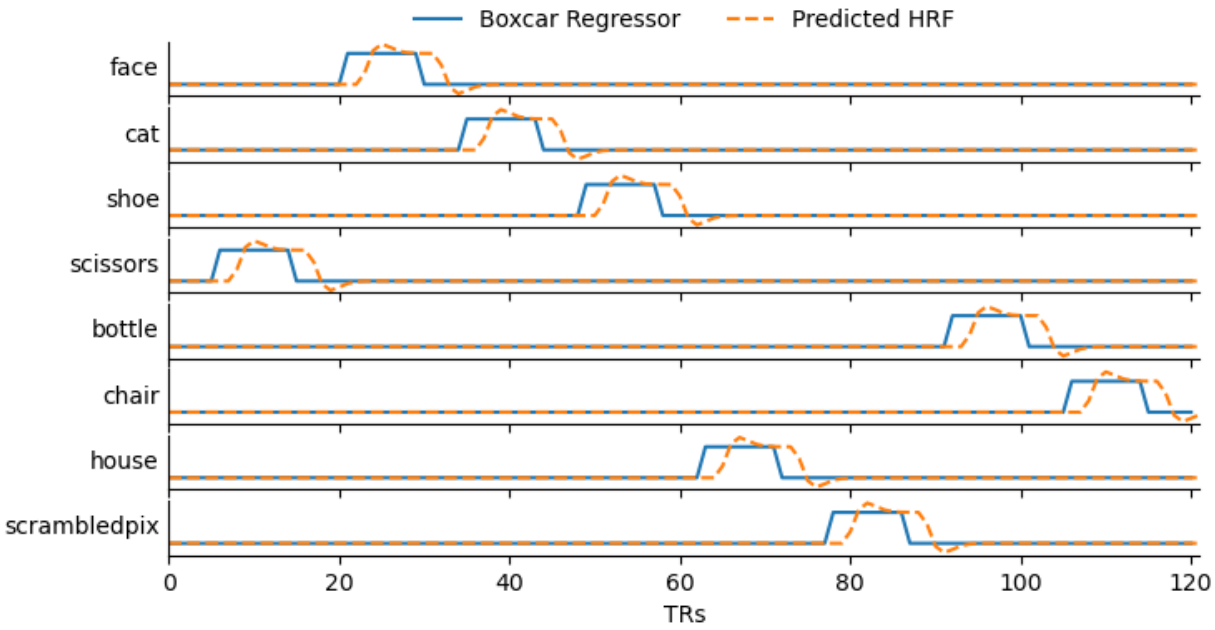
In [ ]:
```
# Load in HRF
from nilearn.glm.first_level import glover_hrf

tr = 2.5
hrf = glover_hrf(tr, oversampling=1, time_length=30)
```

The current boxcar time series capture hypothesized neural activity in response to the stimulus images, but do not reflect the sluggish BOLD response. For now, we'll keep focusing on the first run. In the following cell, convolve each of these boxcar time series with the canonical HRF to construct regressors for the fMRI data.

In [ ]:
```
# Convolve first-run boxcar time series with HRF:
convolved_regressors = np.apply_along_axis(lambda m: np.convolve(m, hrf), axis=
```

Plot the predicted BOLD time series similarly to the boxcar time series above.

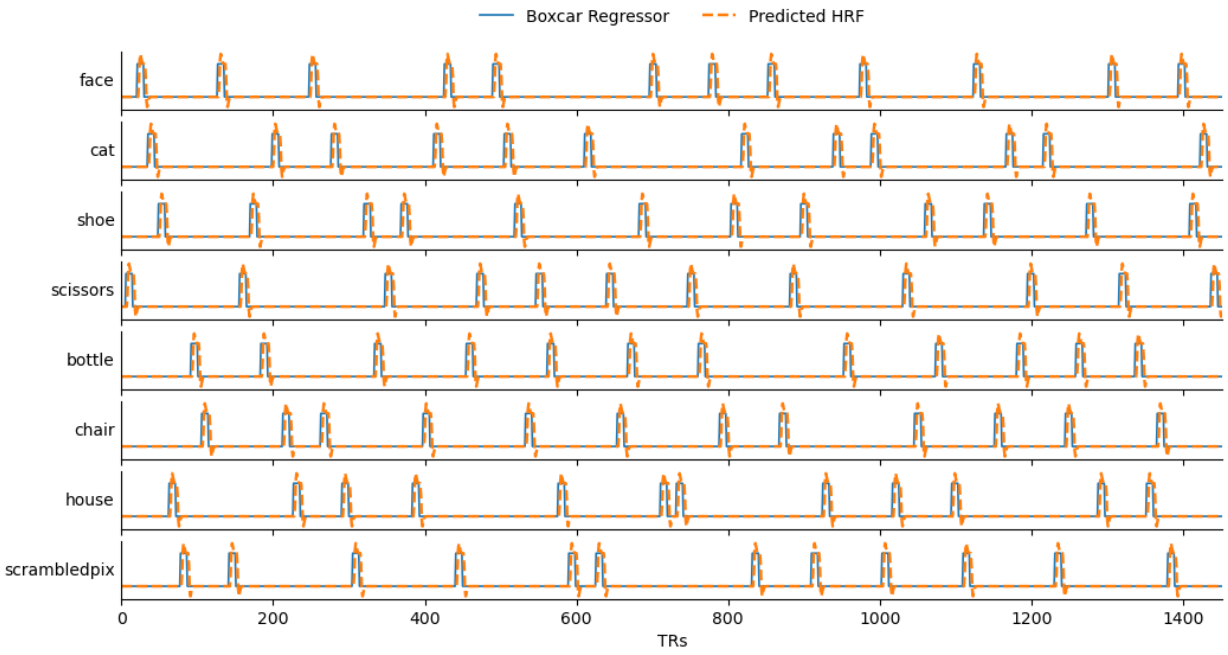
In [ ]:
```
# Plot predicted BOLD time series for first run:
fig, axs = plt.subplots(n_categories, 1, figsize=(8, 4),
                        sharex=True, sharey=True)
for i, (category, regressor, convolved) in enumerate(zip(categories, regressors
    axs[i].plot(regressor, label='Boxcar Regressor')
    axs[i].plot(convolved, '--', label='Predicted HRF')
    axs[i].set(yticks=[])
    axs[i].set_ylabel(category, rotation=0, ha='right', va='center')
    if i == 0:
        axs[i].legend(loc='upper center', frameon=False, ncol=2, bbox_to_anchor
plt.xlabel('TRs')
plt.xlim(0, run_trs)
sns.despine()
```



Now, combining elements of the code from previous cells, repeat this procedure of convolving each within-run boxcar time series with the HRF for all 12 runs. Vertical-stack all of these runwise regressors into a single design matrix `X` with shape `(1452, 8)`. Plot the eight regressors across the entire experiment.

In [ ]:
```
# Loop through runs, apply HRF, and re-stack:
# Construct boxcar time series for the first run
run_id = 0
regressors = []
for category in categories:
    regressor = np.zeros(stimuli.size)
    regressor[np.where(stimuli == category)] = 1
    regressors.append(regressor)
regressors = np.column_stack(regressors)

convolved_regressors = np.apply_along_axis(lambda m: np.convolve(m, hrf), axis=

# Plot regressors across entire experiment:
# Plot boxcar time series for the first run
fig, axs = plt.subplots(n_categories, 1, figsize=(12, 6),
                        sharex=True, sharey=True)
for i, (category, regressor, convolved) in enumerate(zip(categories, regressors
    axs[i].plot(regressor, lw=1.2, label='Boxcar Regressor')
    axs[i].plot(convolved, '--', lw=1.75, label='Predicted HRF')
    axs[i].set(yticks=[])
    axs[i].set_ylabel(category, rotation=0, ha='right', va='center')
    if i == 0:
        axs[i].legend(loc='upper center', frameon=False, ncol=2, bbox_to_anchor
plt.xlabel('TRs')
plt.xlim(0, stimuli.size)
sns.despine()
runs.size
```

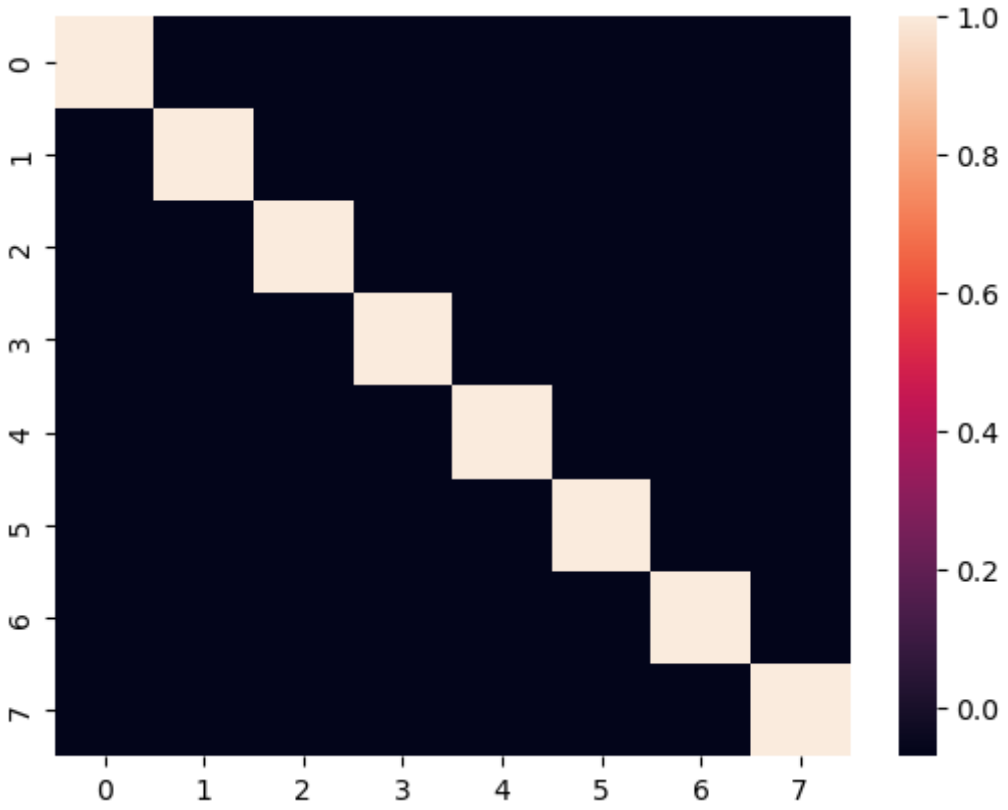
Out[ ]:
```
1452
```



When designing an fMRI experiment, it's important minimize collinearity among your regressors of interest to ensure that your regression model can accurately assign variance to each regressor. One simple way to evaluate collinearity is to compute the pairwise correlations (shaped `(8, 8)`) among your regressors of interest in `X`. Use `np.corrcoef` to compute the pairwise correlations among regressors and plot the resulting correlation matrix; similarly, you could compute the convariance matrix using `np.cov` or `X.T @ X`, or use a more sophisticated metric like variance inflation factor (e.g. `variance_inflation_factor` in `statsmodels.stats.outliers_influence`).

In [ ]:
```
# Compute correlation matrix and plot:
sns.heatmap(np.corrcoef(convolved_regressors.T))
```

Out[ ]:
```
<AxesSubplot:>
```

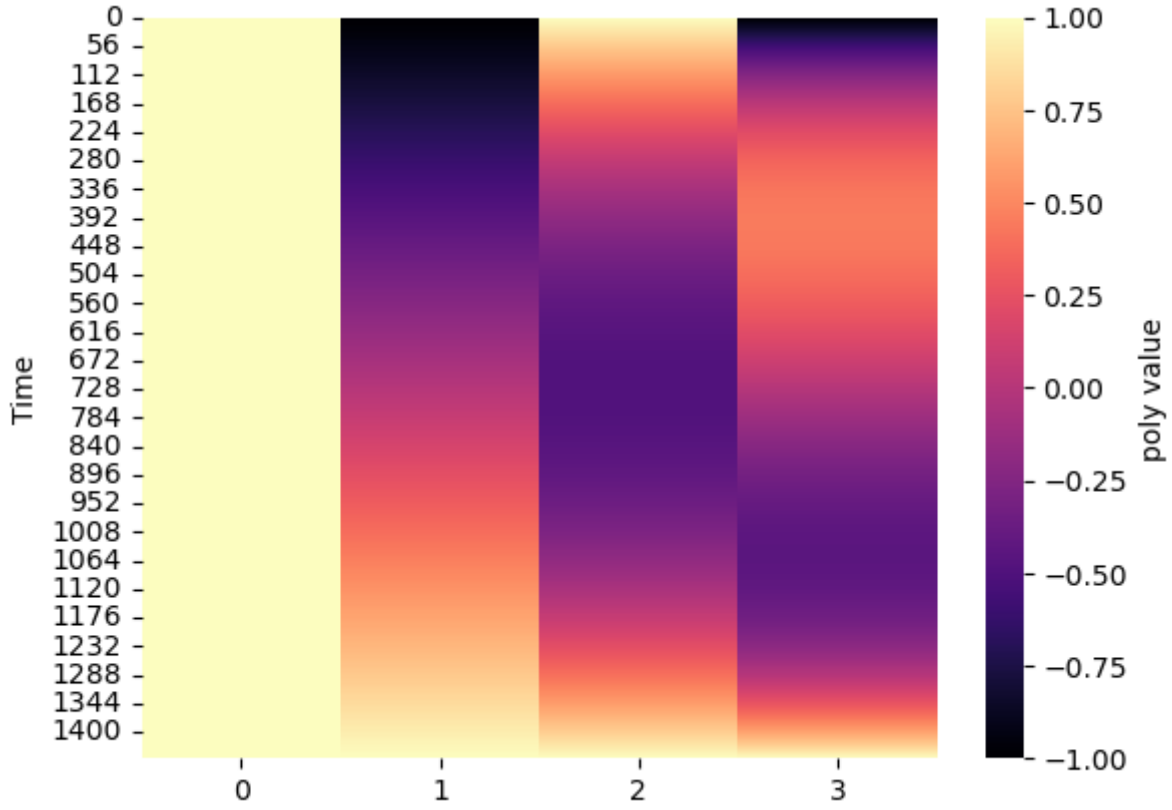


Problem 4: Regression

Now that we have a set of regressors (i.e. predictors) capturing the object categories in our experimental design, we'll use a simple regression analysis to model each voxel time series in the fMRI data. In fMRI analysis, this is often referred to as the mass univariate general linear model or GLM. The goal is to discover which voxels are most responsive to certain object categories such as faces or houses. Recall that the publicly-available Haxby et al., 2001 unfortunately does not include confound variables, so here we'll focus mostly on the regressors of interest. However, we can include 4th-order Legendre polynomials in the same way as above to mitigate any slow drifts.

In [ ]:
```
# Construct 4th-degree Legendre polynomials matching number of TRs:
x_grid = np.linspace(-1, 1, n_trs)
degree = 4

polys = []
for n in np.arange(degree):
    polys.append(eval_legendre(n, x_grid))
polys = np.column_stack(polys)
ax = sns.heatmap(polys, cmap='magma', cbar_kws={'label': 'poly value'})
ax.set_ylabel('Time')
```

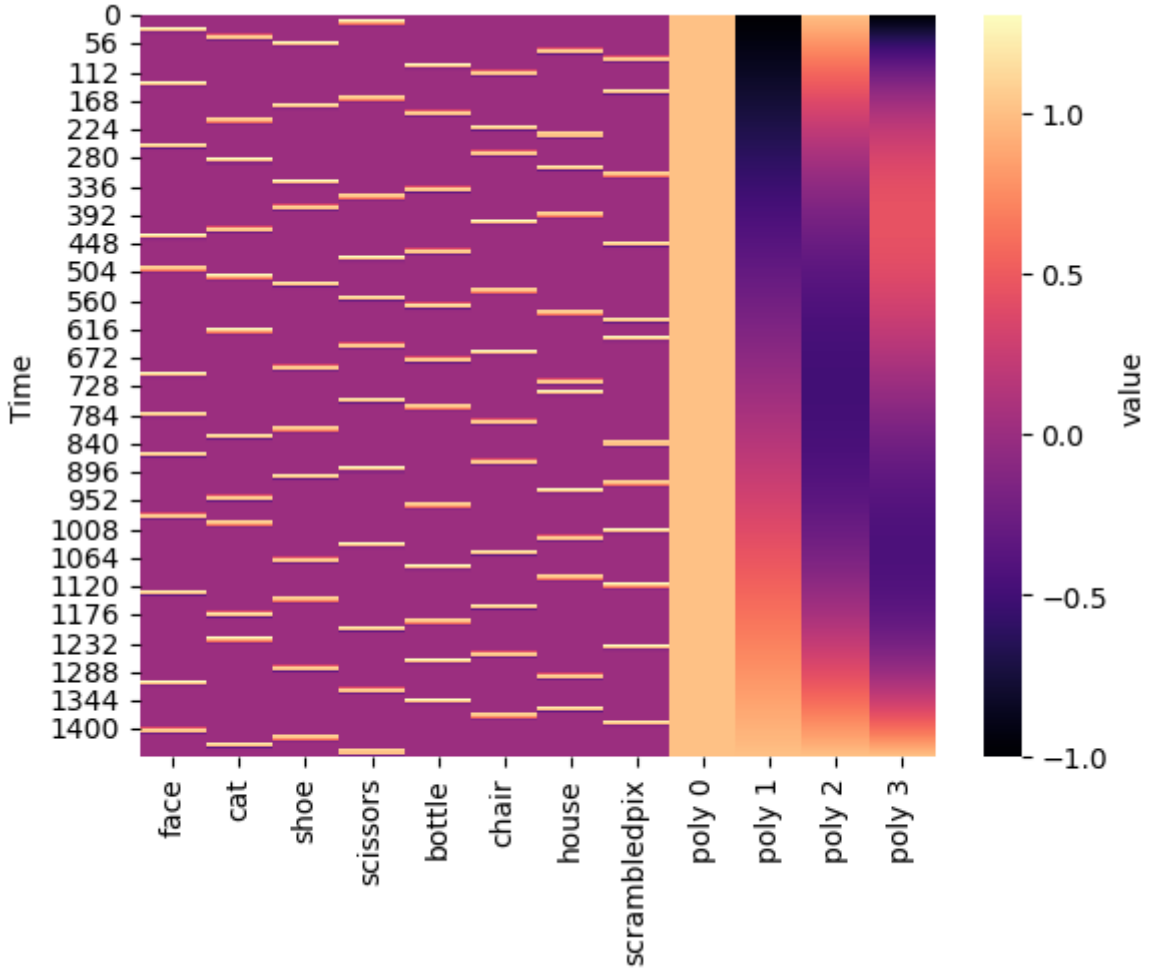
Out[ ]:
```
Text(50.722222222222214, 0.5, 'Time')
```



Stack these polynomial detrending variables alongside the eight category regressors in your design matrix `X`. Keep track of which columns correspond to variables of interest and which correspond to confound variables.

In [ ]:
```
# Stack detrending variables alongside category regressors:
X = np.hstack((convolved_regressors, polys))
X_labels = categories + [f'poly {i}' for i in range(polys.shape[1])]
ax = sns.heatmap(X, cmap='magma', xticklabels=X_labels, cbar_kws={'label': 'val
ax.set_ylabel('Time')
```

Out[ ]:
```
Text(50.722222222222214, 0.5, 'Time')
```



Next, we need to make sure that our fMRI data are prepared. For the sake of computational efficiency, we'll focus on the voxels in VT cortex, rather than running a whole-brain analysis across all voxels. Use the functional data extracted from VT cortex from Part 1. Split the VT data into separate runs, z-score the voxelwise time series within each run, the re-stack the runs.

In [ ]:
```
# Z-score each run and re-stack:
y = np.hstack([zscore(func_mask[:, runs==run], axis=1) for run in np.unique(run
```

Finally, we should be ready to run our regression. First, to keep things as simple as possible, we'll run the regression using `lstsq` from `np.linalg`. Use `lstsq` to compute the regression coefficients (i.e. "beta values") for the eight category regressors.

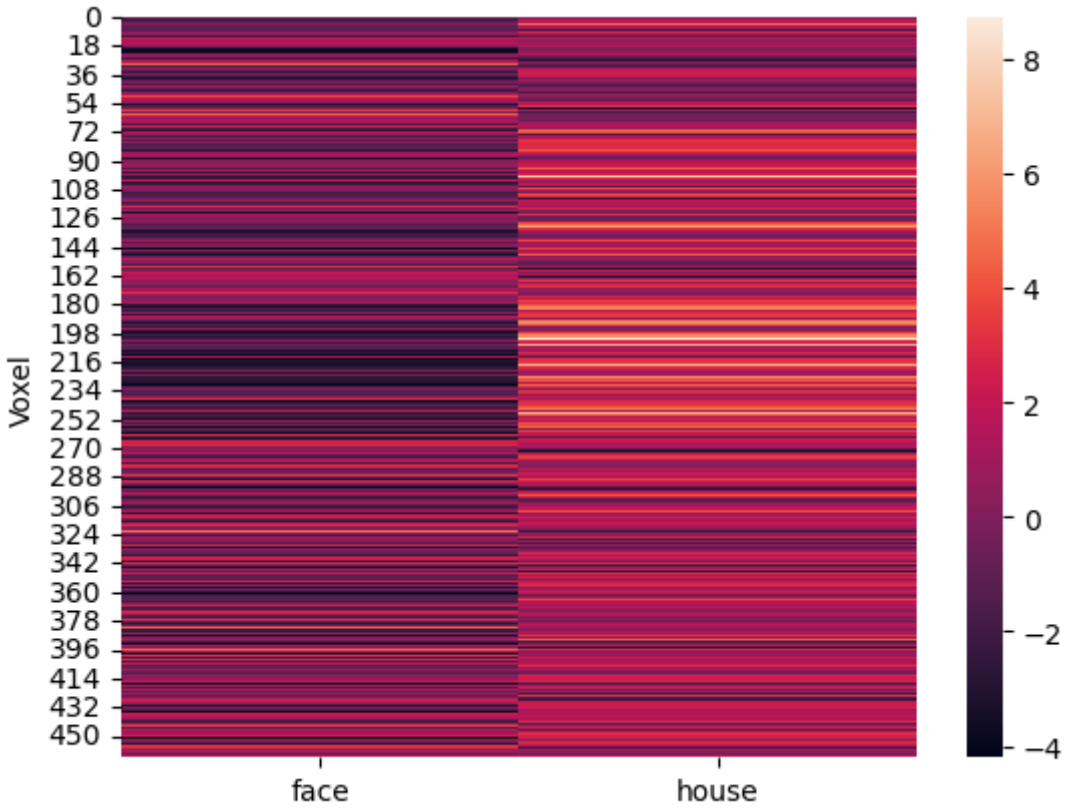
In [ ]:
```
# Use least-squares to compute betas:
betas, _, _, _ = np.linalg.lstsq(X, y.T)
betas.shape
```

```
/tmp/ipykernel_201013/523679018.py:2: FutureWarning: `rcond` parameter will ch
ange to the default of machine precision times ``max(M, N)`` where M and N are
the input matrix dimensions.
To use the future default and silence this warning we advise to pass `rcond=No
ne`, to keep using the old, explicitly pass `rcond=-1`.
  betas, _, _, _ = np.linalg.lstsq(X, y.T)
```

Out[ ]: (12, 464)

Now, use a contrast vector to perform two comparisons: (1) faces > all seven other categories, and (2) houses > all seven other categories. Remember, these contrast vectors should sum to zero. Multiple the contrast vector by the corresponding betas and sum across betas to compute the contrast; each comparison should result in a single map for VT cortex.

In [ ]:
```
betas_cat = betas[:8]
betas_cat_labels = X_labels[:8]

# Create contrast vectors:
contrast_face_vec = np.ones(len(betas_cat_labels)) * -1
contrast_face_vec[betas_cat_labels.index('face')] = 7
contrast_house_vec = np.ones(len(betas_cat_labels)) * -1
contrast_house_vec[betas_cat_labels.index('house')] = 7
# Apply contrast vectors to betas:
contrast_face = betas_cat.T @ contrast_face_vec
contrast_house = betas_cat.T @ contrast_house_vec
sns.heatmap(np.hstack((contrast_face[:, np.newaxis], contrast_house[:, np.newax
plt.ylabel('Voxel');
```

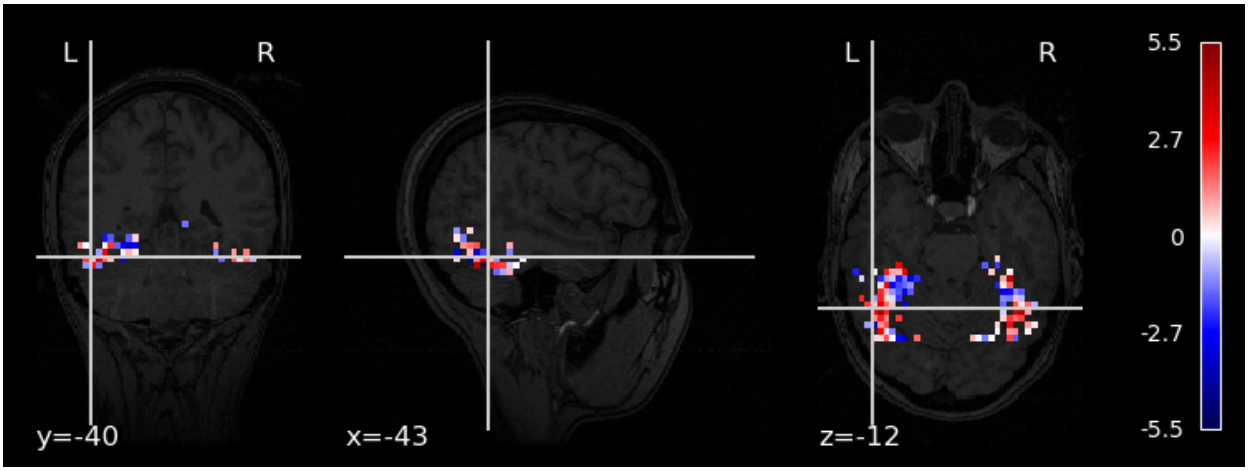


Finally, reinsert these beta values back into a full NIfTI image and use `plot_stat_map` from `nilearn.plotting` to visualize the resulting maps for the faces-vs-all contrast and the houses-vs-all contrast. Try using the T1-weighted anatomical image as the `bg_img`, then pick a good set of coordinates for visualizing VT (e.g. `cut_coords = (-43, -40, -12)`). What do you notice about the anatomical localization of face-responsive versus house-responsive areas of VT cortex?

> They appear to be somewhat anticorelated.

In [ ]:
```
from nilearn.plotting import plot_stat_map
img = nib.load(haxby_dataset['func'][0])
# Reinsert VT face contrast map into full NIfTI image:
contrast_face_full = np.zeros(func_data.shape[:3])
contrast_face_full[vt_mask_idx] = contrast_face
vt_face_contrast = nib.Nifti1Image(contrast_face_full, affine=img.affine)

# Plot face-vs-all contrast:
plot_stat_map(vt_face_contrast, bg_img=haxby_dataset['anat'][0], cut_coords=(-4
```

Out[ ]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7f1136025160>
```



In [ ]:
```
# Reinsert VT house contrast map into full NIfTI image:
img = nib.load(haxby_dataset['func'][0])
# Reinsert VT face contrast map into full NIfTI image:
contrast_house_full = np.zeros(func_data.shape[:3])
contrast_house_full[vt_mask_idx] = contrast_house
vt_house_contrast = nib.Nifti1Image(contrast_house_full, affine=img.affine)

# Plot hosue-vs-all contrast:
plot_stat_map(vt_house_contrast, bg_img=haxby_dataset['anat'][0], cut_coords=(-
```

Out[ ]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7f1135f901f0>
```