NEU502B Homework 4: Multivariate pattern analysis

Due March 27, 2024

Submission instructions: First, rename your homework notebook to include your name (e.g. `homework-4-nastase.ipynb`); keep your homework notebook in the `homework` directory of your clone of the class repository. Prior to submitting, restart the kernel and run all cells (see *Kernel* > *Restart Kernel and Run All Cells...*) to make sure your code runs and the figures render properly. Only include cells with necessary code or answers; don't include extra cells used for troubleshooting. To submit, `git add`, `git commit`, and `git push` your homework to your fork of the class repository, then make a pull request on GitHub to sync your homework into the class repository.

In this homework assignment, you will work through three commonly used methods in cognitive computational neuroscience: (1) neural decoding via multivariate pattern analysis (MVPA); (2) representational similarity analysis (RSA); and (3) voxelwise encoding analysis using regularized regression. Each of these problems builds on tools and ideas we've introduced in the in-class lab notebooks.

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

Problem 1: Multivariate pattern classification

First, we'll start with a simple example of classifying distributed response patterns for different object categories from Haxby et al., 2001. We'll begin by loading in the data, as well as labels for the stimuli and runs. 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.

```
from nilearn import datasets
from nilearn.image import index_img
import pandas as pd

# Change this path to a directory on your computer!
data_dir = '/Users/polina/neu502b-2024/nilearn-data'

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

# Load in session metadata as pandas DataFrame
session = pd.read_csv(haxby_dataset.session_target[0], sep=" ")

# Extract stimuli and run labels for this subject
```

```
stimuli, runs = session['labels'].values, session['chunks'].values

# Create a boolean array indexing TRs containing a stimulus (non-rest)
task_trs = stimuli != 'rest'

# Get list of unique stimulus categories (excluding rest)
categories = [c for c in np.unique(stimuli) if c != 'rest']

# Extract task TRs for fMRI data and stimulus/run labels
func_task = index_img(haxby_dataset.func[0], task_trs)
stimuli_task = stimuli[task_trs]
runs_task = runs[task_trs]
```

Use `NiftiMasker` (with `standardize=True`) to create a masker for ventral temporal (VT) cortex. Use the masker to extract the the NumPy array containing the functional data. (We'll analyze the data using scikit-learn rather than nilearn.)

```
# Get the VT mask file and creater masker:
from nilearn.maskers import NiftiMasker

mask_vt = haxby_dataset.mask_vt[0]
masker = NiftiMasker(mask_img = mask_vt, standardize= True)


# Use masker to extract numpy array for VT:
func_task = index_img(haxby_dataset.func[0], task_trs)
vt_masked_data = masker.fit_transform(func_task)
```

```
/Users/polina/anaconda3/envs/neu502b/lib/python3.11/site-packages/
nilearn/image/resampling.py:492: UserWarning: The provided image has
no sform in its header. Please check the provided file. Results may
not be as expected.
  warnings.warn(
```

Now, we'll set up a full SVM classification analysis using leave-one-run-out outer cross-validation with a nested leave-one-run-out inner cross-validation loop for grid search across the values of the SVM regularization parameter C. Sounds like a lot! But scikit-learn makes it pretty straightforward. First, initialize the `LinearSVC` estimator. Since this well-behaved dataset has the same number of samples for each stimulus category in each run, we can perform leave-one-run-out cross-validation using just `KFold` rather than having to specify the runs directly. Initalize an outer `KFold` cross-validator with 12 splits and an inner `KFold` cross-validator with 11 splits. We'll search over a handful of C parameters: `param_grid = {'C': [1e-2, 1e-1, 1]}`. Initialize the `GridSearchCV` estimator with the SVM estimator, the parameter grid, and the inner cross-validator; then, submit this estimator to `cross_val_predict` with the outer cross-validator to run the full analysis. (This may take a few minutes to run!)

```
# Suppress some warnings (e.g. SVM convergence) just to clean up
output
import warnings
```

```python
from sklearn.model_selection import (cross_val_predict,
                                     GridSearchCV,
                                     KFold)
from sklearn.svm import LinearSVC
warnings.filterwarnings("ignore")

# Initialize SVM and outer/inner CVs:
classifier = LinearSVC()
outer_cv = KFold(n_splits=12)
inner_cv = KFold(n_splits=11)

# Set up parameter grid:
param_grid = {'C': [1e-2, 1e-1, 1]}

# Initialize GridSearchCV estimator:
gse = GridSearchCV(classifier, param_grid, cv = inner_cv)

# Generate predictions using cross_val_predict:
y_pred = cross_val_predict(gse,
                           vt_masked_data,
                           stimuli_task,
                           cv=outer_cv)
```

Inspect the resulting predictions. We'll evaluate our classifier's predictions in two ways. First, use `accuracy_score` from `sklearn.metrics` to evaluate the predictions (across all test sets) against the actual labels in terms of a single classification accuracy. Procedurally, this is slightly different from computing accuracies on the test for each fold and averaging them—but the resulting value should be the same.

```python
# Print accuracy score:
from sklearn.metrics import accuracy_score

acc = accuracy_score(stimuli_task, y_pred)
print(f"Classification accuracy= {acc}")
```

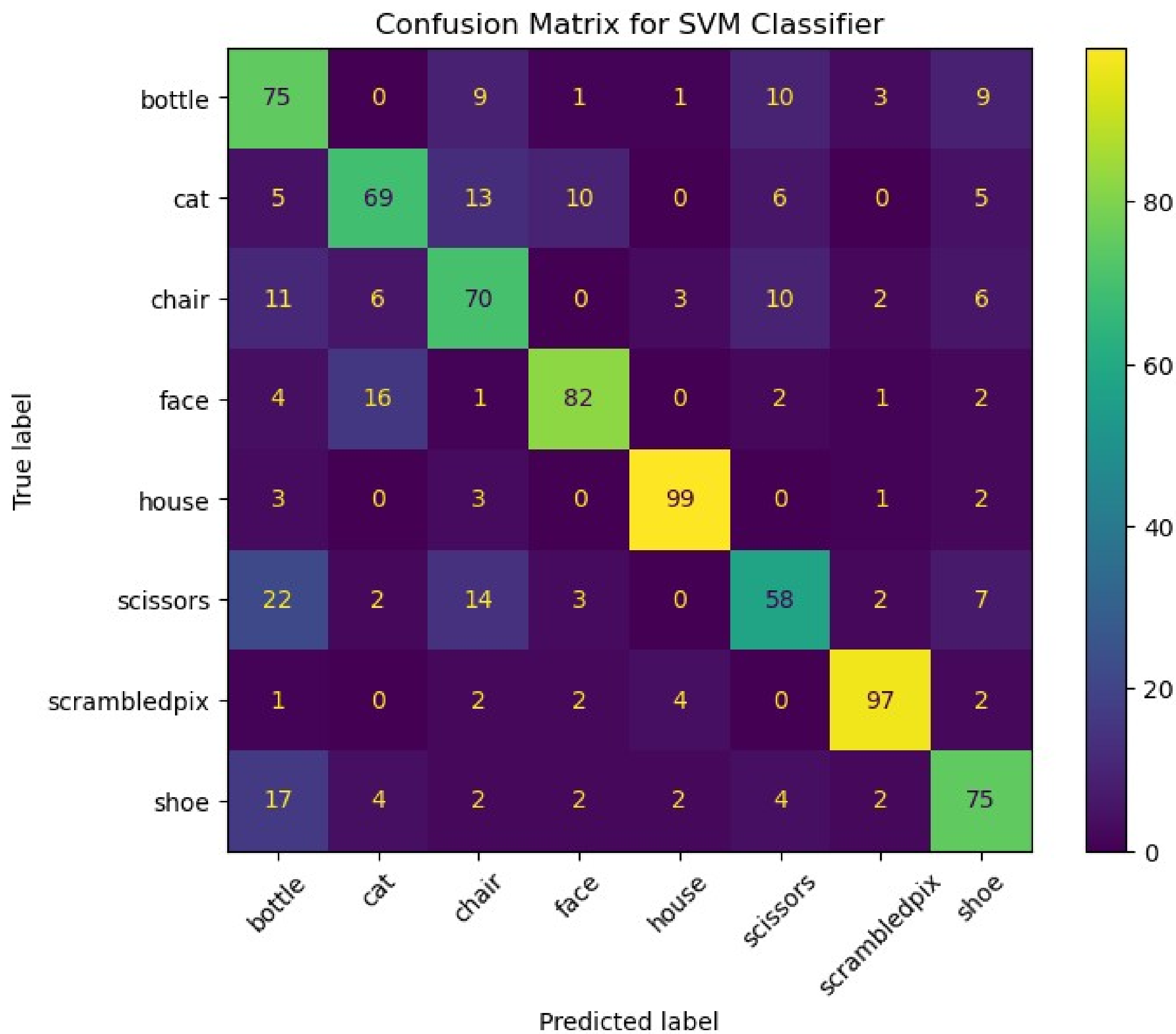
```
Classification accuracy= 0.7233796296296297
```

To better understand what our classifer is doing (i.e. what it's getting right and what it's getting wrong), we'll construct a confusion matrix. Construct the confusion matrix from the actual stimulus labels and the classifer's predicted labels and plot it below. What categories does the classifier tend to misclassify?

The classifier does pretty well overall, but slightly tends to misclassify scissors (sometimes conflating them with other objects like bottles and chairs), cat (conflating it with chair or face), and chair (conflating it with other objects like bottle and scissors).

```python
# Create confusion matrix from true and predicted labels:
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
cm = confusion_matrix(stimuli_task, y_pred)

# Plot confusion matrix:
fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay(cm, display_labels=categories).plot(ax=ax)
ax.tick_params(axis='x', labelrotation=45)
plt.title("Confusion Matrix for SVM Classifier");
```

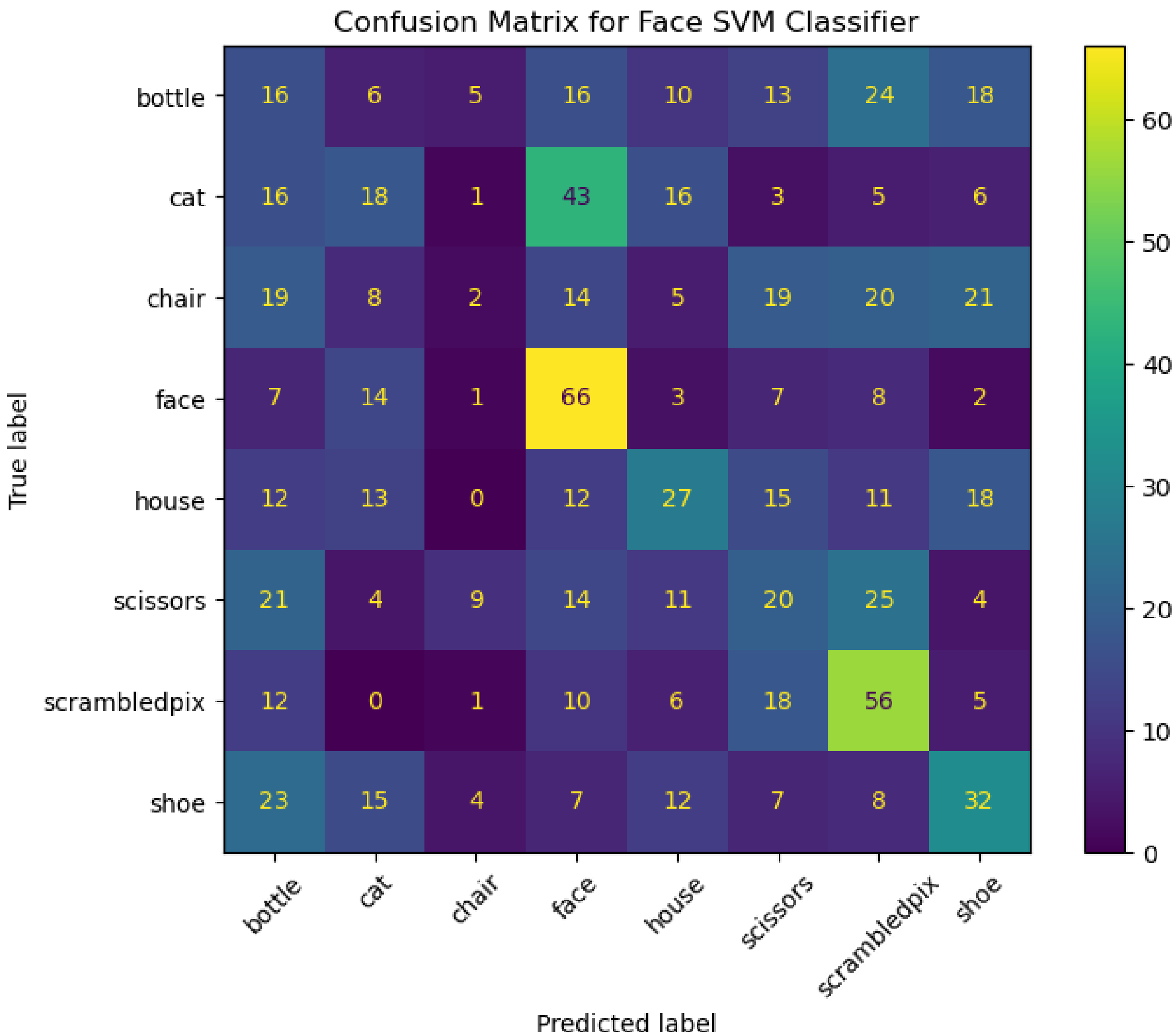


Lastly, we'll repeat the same analysis for functional regions of interest (ROIs) maximally responsive to faces (roughly FFA) and houses (roughly PPA). Use the `mask_face` and `mask_house` files from the dataset to create an FFA masker and a PPA masker; extract the functional data for both. Submit these datasets to the same analysis as above, and visualize the results in terms of an overall accuracy score and confusion matrix. Interpret the accuracies and confusion matrices in light of the expected chance accuracy, given what you know about these ROIs.

The expected chance accuracy is 1/n, which is 0.125 for these ROIs. Both FFA and PPA had accuracies above chance, which is good, but not as high as the overall SVM classifier from before. The confusion matrices show that the face SVM most accurately identifies faces (which is expected) and scrambled pixels, but has a slight tendency to classify cats as faces (which might be because cats have faces). The house SVM performs better and is most accurately able to identify houses, as well as (with a slightly lesser accuracy) faces, scrambled pixels, and chairs. This might make sense when considering that faces and chairs are often elements of places, so PPA might have some sort of distinct responses to these categories as well.

```
# Create masker for FFA:
mask_face = haxby_dataset.mask_face[0]
masker_face = NiftiMasker(mask_img = mask_face, standardize= True)

# Create masker for PPA:
mask_house = haxby_dataset.mask_house[0]
masker_house = NiftiMasker(mask_img = mask_house, standardize= True)

# Uses masker to extract numpy array for VT:
func_task = index_img(haxby_dataset.func[0], task_trs)

face_masked_data = masker_face.fit_transform(func_task)
house_masked_data = masker_house.fit_transform(func_task)

# Initialize SVM and outer/inner CVs:
classifier = LinearSVC()
outer_cv = KFold(n_splits=12)
inner_cv = KFold(n_splits=11)

# Set up parameter grid:
param_grid = {'C': [1e-2, 1e-1, 1]}

# Initialize GridSearchCV estimator:
gse = GridSearchCV(classifier, param_grid, cv = inner_cv)

# Generate predictions using cross_val_predict:
y_pred = cross_val_predict(gse,
                           face_masked_data,
                           stimuli_task,
                           cv=outer_cv)

# Print accuracy score and plot confusion matrix:
acc = accuracy_score(stimuli_task, y_pred)
print(f"Classification accuracy= {acc}\n\n")

cm = confusion_matrix(stimuli_task, y_pred)

fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay(cm, display_labels=categories).plot(ax=ax)
ax.tick_params(axis='x', labelrotation=45)
plt.title("Confusion Matrix for Face SVM Classifier");
```

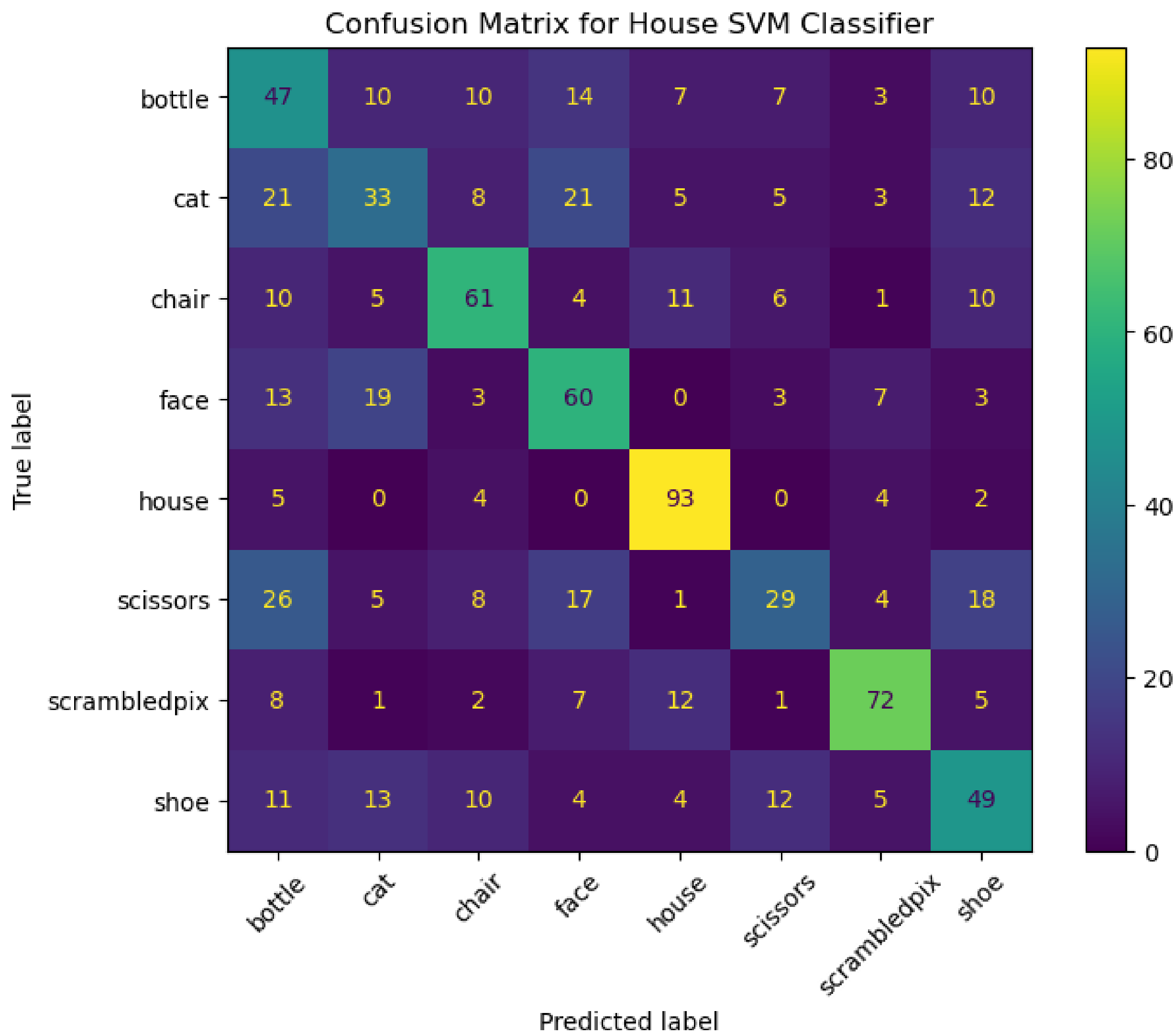
```
Classification accuracy= 0.2743055555555556
```




```
# Initialize SVM and outer/inner CVs:
classifier = LinearSVC()
outer_cv = KFold(n_splits=12)
inner_cv = KFold(n_splits=11)

# Set up parameter grid:
param_grid = {'C': [1e-2, 1e-1, 1]}

# Initialize GridSearchCV estimator:
gse = GridSearchCV(classifier, param_grid, cv = inner_cv)

# Generate predictions using cross_val_predict:
```

```python
y_pred = cross_val_predict(gse,
                           house_masked_data,
                           stimuli_task,
                           cv=outer_cv)

# Print accuracy score and plot confusion matrix:
acc = accuracy_score(stimuli_task, y_pred)
print(f"Classification accuracy= {acc}\n\n")

cm = confusion_matrix(stimuli_task, y_pred)

fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay(cm, display_labels=categories).plot(ax=ax)
ax.tick_params(axis='x', labelrotation=45)
plt.title("Confusion Matrix for House SVM Classifier");
```

```
Classification accuracy= 0.5138888888888888
```



Problem 2: Representational similarity analysis

In this problem, we'll apply representational similarity analysis (RSA) to the human fMRI dataset from Kriegeskorte et al., 2008. We'll begin by loading in the ROI data and labels.

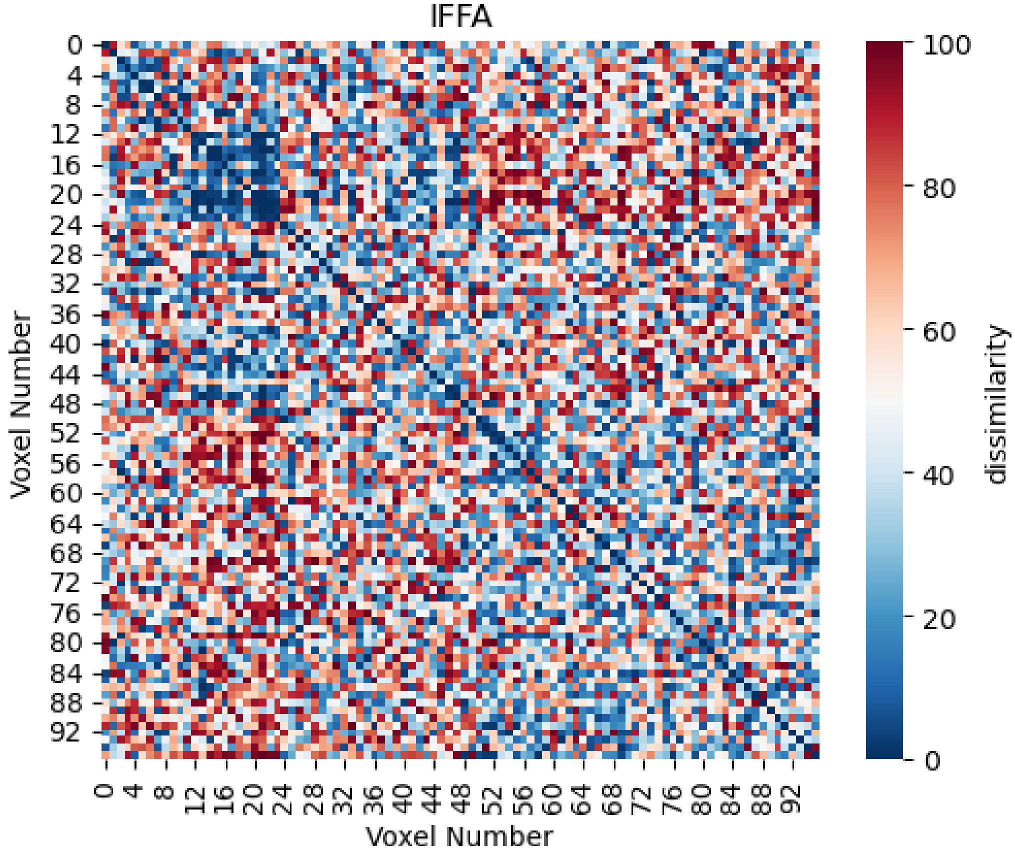
```
# Load in Kriegekorte dataset and labels
kriegeskorte_dataset = dict(np.load('kriegeskorte_dataset.npz',
                                    allow_pickle=True))

roi_data = kriegeskorte_dataset['roi_data'].item()
category_names = kriegeskorte_dataset['category_names']
category_labels = kriegeskorte_dataset['category_labels']
images = kriegeskorte_dataset['images']
subject_labels = ['KO', 'SN', 'TI']
roi_labels = ['lFFA', 'rFFA', 'lPPA', 'rPPA']
```

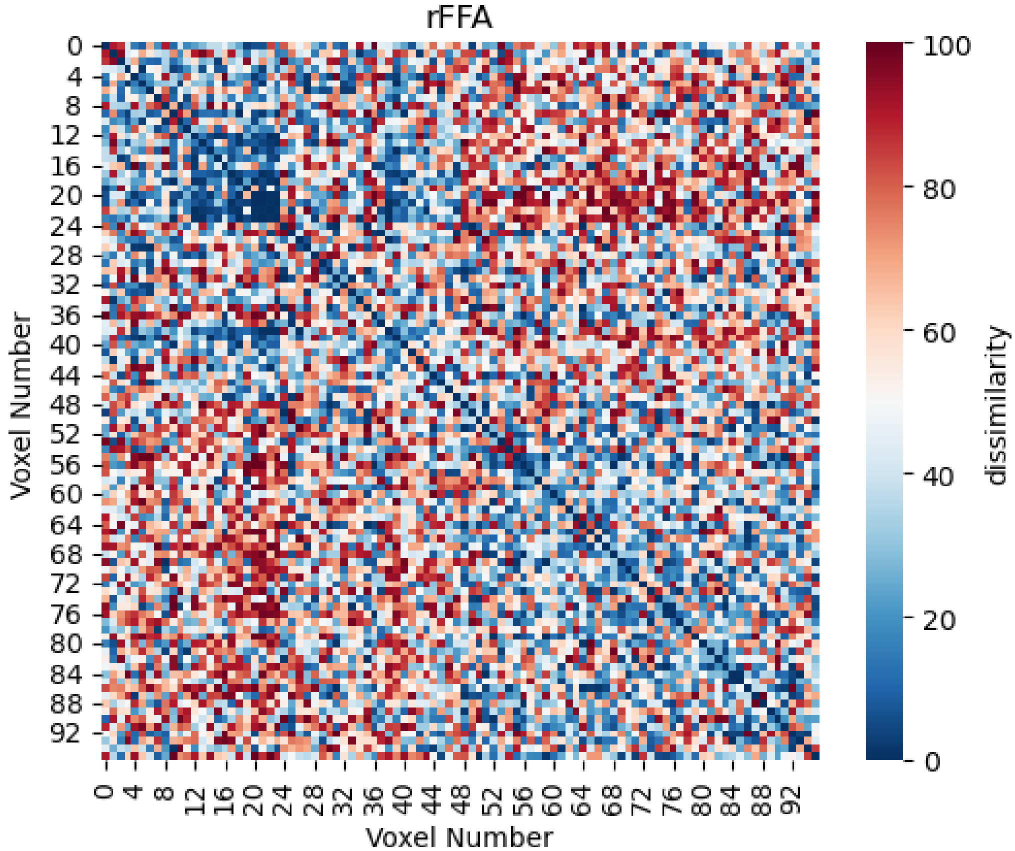
We provide a `rank_percentile` function for visualizing RDMs in a way that more closely matches the paper.

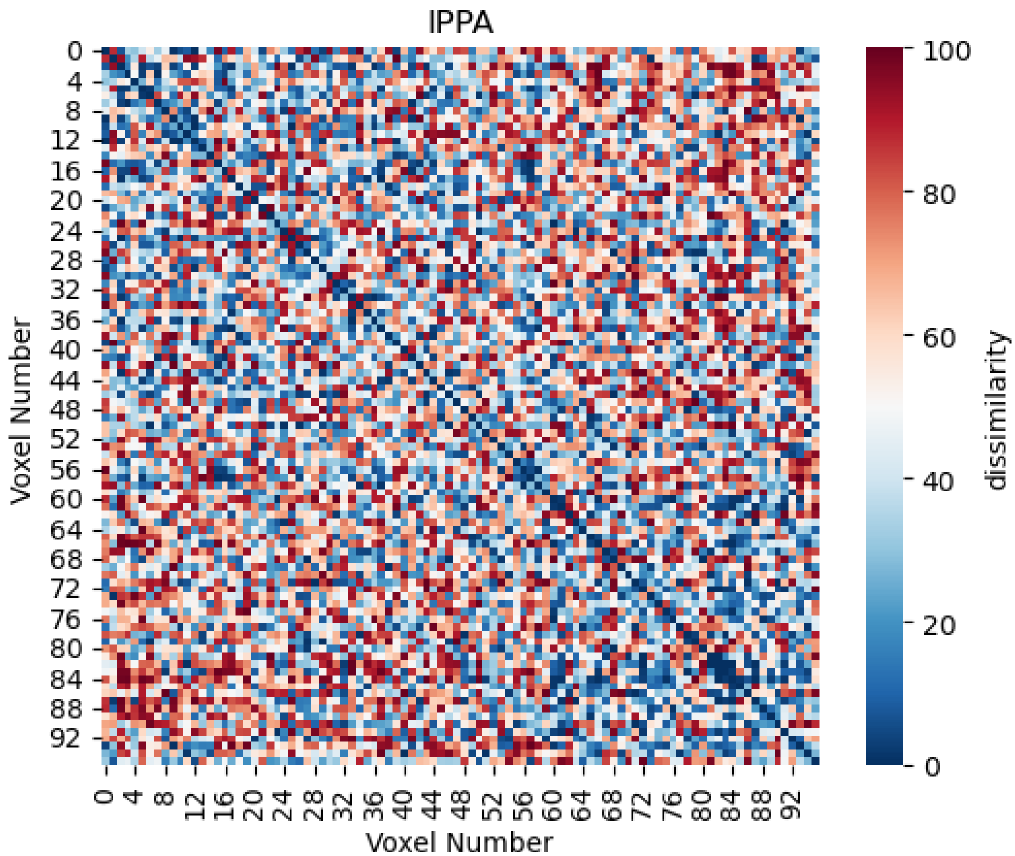
```
from scipy.stats import rankdata

def rank_percentile(a):
    return rankdata(a) / len(a) * 100
```

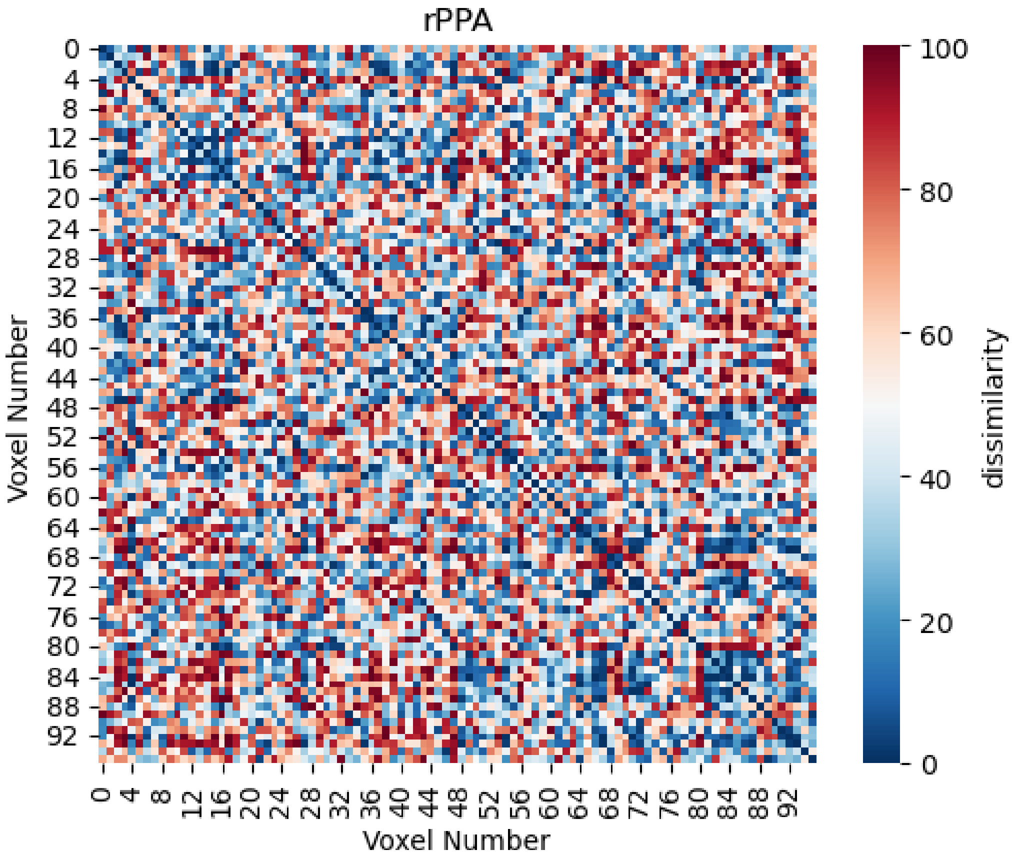
First, compute RDMs for the `'lFFA'`, `'rFFA'`, `'lPPA'`, and `'rPPA'` ROIs for subject `'TI'` using correlation distance. Here, we recommend z-soring each voxel across samples prior to computing the pairwise dissimilarities. Plot the RDMs for each ROI using the `rank_percentile` function provided above.

```
# Plot ROI RDMs for subject TI:
from scipy.stats import zscore
from scipy.spatial.distance import pdist, squareform

rdm_lFFA = rank_percentile(pdist(zscore(roi_data['TI']['lFFA'],
axis=0), metric='correlation'))
rdm_rFFA = rank_percentile(pdist(zscore(roi_data['TI']['rFFA'],
axis=0), metric='correlation'))
rdm_lPPA = rank_percentile(pdist(zscore(roi_data['TI']['lPPA'],
axis=0), metric='correlation'))
rdm_rPPA = rank_percentile(pdist(zscore(roi_data['TI']['rPPA'],
axis=0), metric='correlation'))

all_rdms = [rdm_lFFA, rdm_rFFA, rdm_lPPA, rdm_rPPA]
for i in range(len(all_rdms)):
    sns.heatmap(squareform(all_rdms[i]),
                square=True,
                cmap='RdBu_r',
                cbar_kws={'label': 'dissimilarity'})
    plt.title(f"{roi_labels[i]}")
    plt.xlabel("Voxel Number")
    plt.ylabel("Voxel Number")
    plt.show()
```


IFFA
Voxel Number
Voxel Number
dissimilarity
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
68
72
76
80
84
88
92
100
80
60
40
20
0


rFFA
Voxel Number
Voxel Number
dissimilarity
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
68
72
76
80
84
88
92
100
80
60
40
20
0

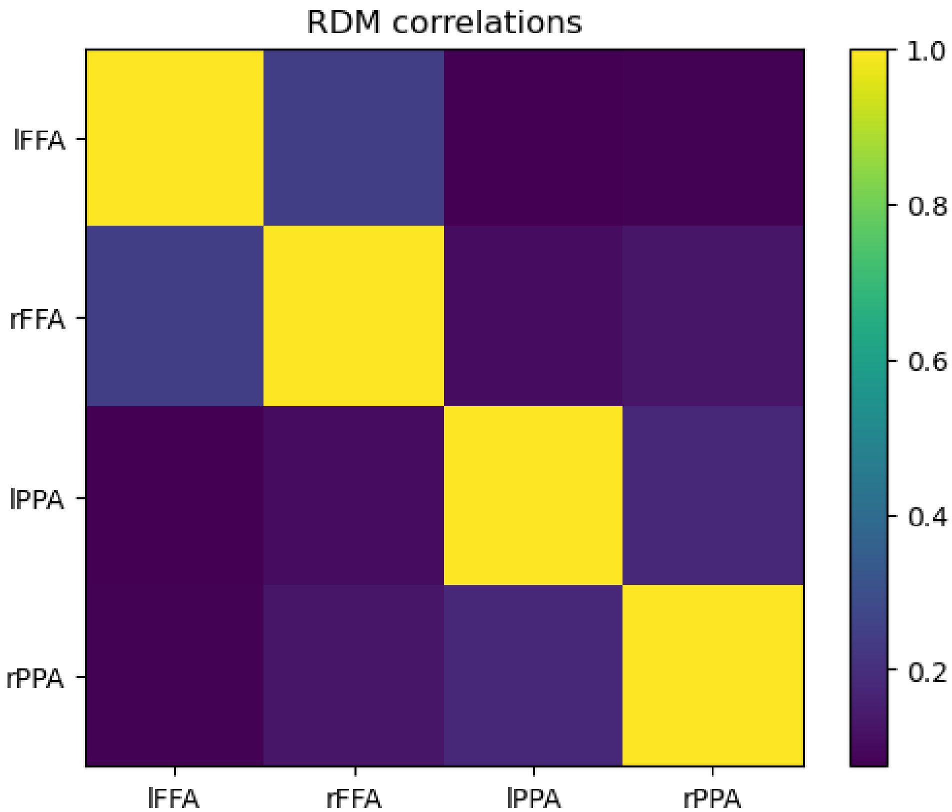

IPPA
Voxel Number
Voxel Number
dissimilarity
0
20
40
60
80
100



RSA allows us to compare the representational geometries of different ROIs. Compute the correlation between each pair of the four ROIs. Plot this similarity matrix. Which ROIs have the most similar representational geometries?

Right and left FFA have the most similar representational geometries to each other, followed by left and right PPA.

```
# Compute correlations between each pair or ROI RDMs:
all_rdms = [rdm_lFFA, rdm_rFFA, rdm_lPPA, rdm_rPPA]
rdm_cross_corr = np.corrcoef(all_rdms)

# Plot correlation matrix:
plt.imshow(rdm_cross_corr);
plt.xticks([0, 1, 2, 3], roi_labels)
plt.yticks([0, 1, 2, 3], roi_labels)
plt.title("RDM correlations")
plt.colorbar();
```



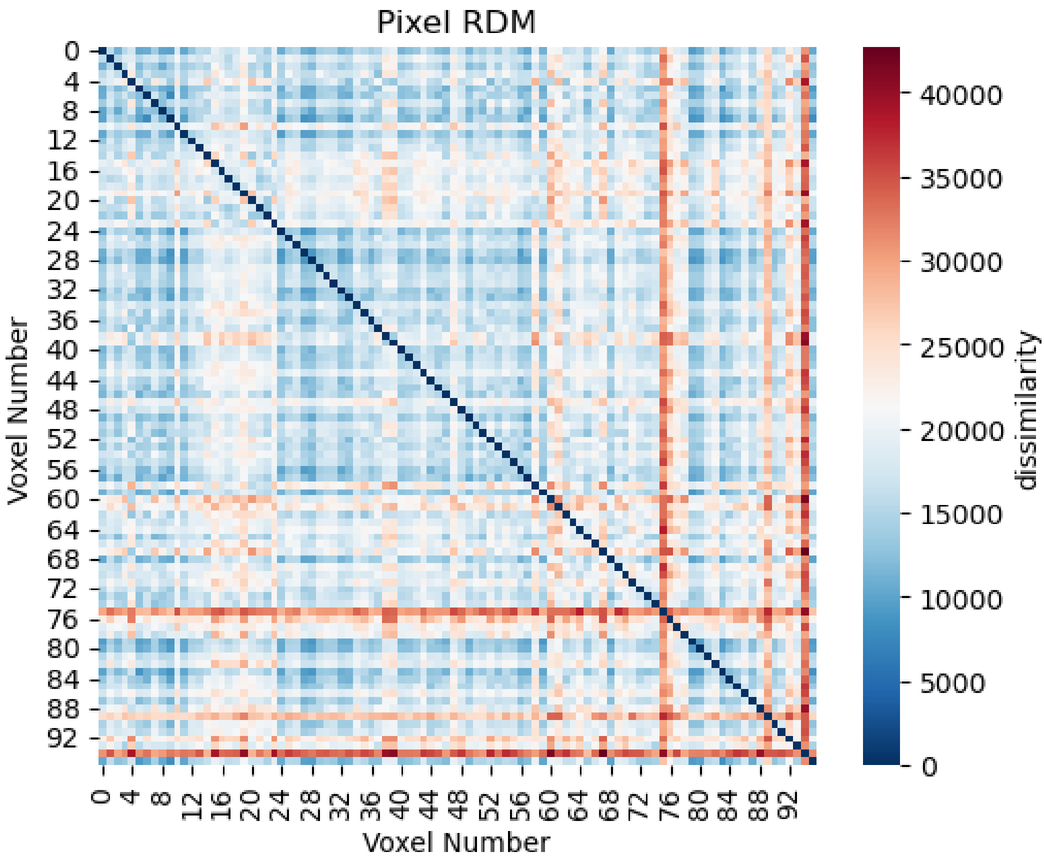
Stack all four ROIs to create a single combined ROI for each subject `'SN'` and `'TI'`. What is the Spearman correlation between `'SN'`'s and `'TI'`'s representational geometries?

```
# Combine SN and TI ROIs into single VT ROI and compute RDMs:
from scipy.stats import spearmanr

# TI
TI_concat = np.concatenate([roi_data["TI"]["lFFA"], roi_data["TI"]
["rFFA"], roi_data["TI"]["lPPA"], roi_data["TI"]["rPPA"]], axis=1)

rdm_TI_concat = rank_percentile(pdist(zscore(TI_concat, axis=0),
metric='correlation'))

# SN
SN_concat = np.concatenate([roi_data["SN"]["lFFA"], roi_data["SN"]
["rFFA"], roi_data["SN"]["lPPA"], roi_data["SN"]["rPPA"]], axis=1)

rdm_SN_concat = rank_percentile(pdist(zscore(SN_concat, axis=0),
metric='correlation'))


# Compute correlations between SN and TI's VT RDMs:
corr = spearmanr(rdm_TI_concat, rdm_SN_concat)[0]
print(f"Spearman correlation = {corr}")
```

```
Spearman correlation = 0.394897619062583
```

We can test different "model" RDMs according to how well they approximate a given neural. Here, for the sake of brevity, we'll construct an extremely simple RDM capturing low-level visual structure. Flatten each image file into a one-dimensional array of pixel values (across three color channels). Next, compute the pairwise Euclidean distances between these image vectors to construct an RDM capture low-level visual similarities. Plot this pixel RDM and compute it's Spearman correlation with `'TI'`s VT RDM?

```python
# Create a pixel-based RDM:
flat_imgs = []
for img in images:
    flat_imgs.append(img.flatten())

flat_imgs = np.array(flat_imgs)

rdm_ti_euc = pdist(flat_imgs, metric='euclidean')

sns.heatmap(squareform(rdm_ti_euc),
        square=True,
        cmap='RdBu_r',
        cbar_kws={'label': 'dissimilarity'})
plt.title("Pixel RDM")
plt.xlabel("Voxel Number")
plt.ylabel("Voxel Number")
plt.show()

# Compute correlations with TI's VT RDM:
corr = spearmanr(rdm_TI_concat, rdm_ti_euc)[0]
print(f"\n\nSpearman correlation = {corr}")
```




```
Spearman correlation = 0.021796286291858 63
```

Problem 3: Voxelwise encoding analysis

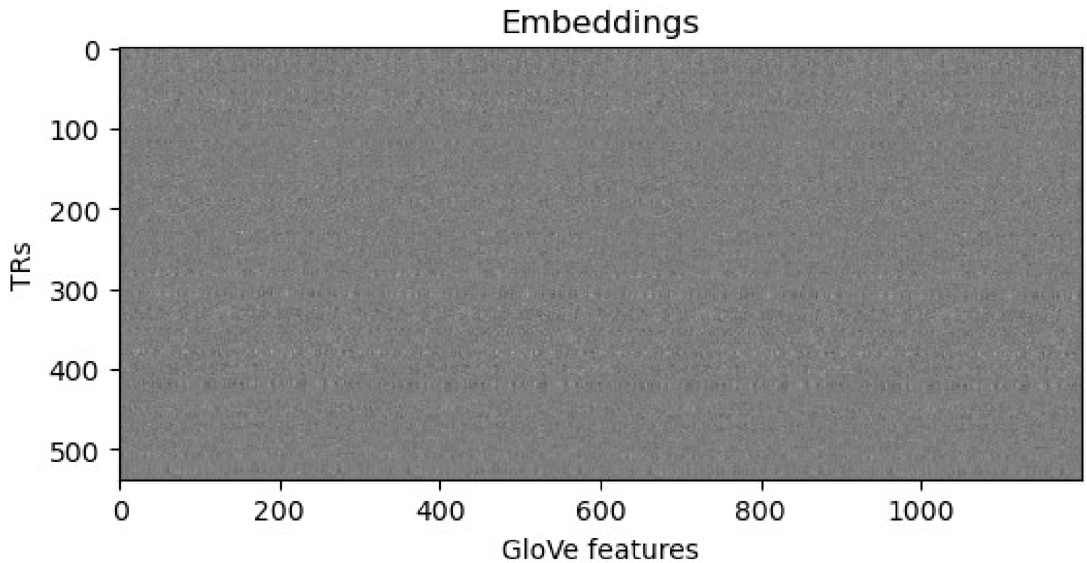
In this problem, we'll return to *encoding analysis*, using regularized regression and out-of-sample prediction in individual voxels. We will use word embeddings derived from the natural language processing (NLP) model GloVe to map semantic encoding onto the brain. You can simply load the `story_transcript.txt` file in a text editor to visualize the transcript for the spoken story by Carol Daniel. Each line of this file corresponds to a TR in the fMRI data. Next, we extracted word embeddings from GloVe for each word in each TR. For TRs containing multiple words, we averaged the embeddings. Finally, we horizontally stacked the embeddings at lags of 2, 3, 4, and 5 TRs (3, 4.5, 6, and 7.5 seconds relative to word onset) to account variable hemodynamic lags (this is effectively a finite impulse response model). Inspect and interpret the shape of the word embeddings, and visualize this matrix.

`embeddings` is 538 TRs (samples) by 1200 GloVe model features.

```
# Load and visualize word embeddings:
embeddings = np.load("story_embeddings.npy")
print(f"embeddings shape: {embeddings.shape}")
```

```
plt.imshow(zscore(embeddings, axis=0), cmap="binary_r")
plt.xlabel("GloVe features")
plt.ylabel("TRs")
plt.title("Embeddings");
```

```
embeddings shape: (538, 1200)
```



We used fMRI to measure a subject's brain activity while they listened to the spoken story. Here, to reduce computational demands, we have spatially downsampled the fMRI data using an atlas containing 400 parcels. That is, for each parcel, we averaged the voxel time series within that parcel. Rather than fitting encoding models to tens of thousands of voxels, we'll fit our encoding model to each of the 400 parcels. Load in the `story_parcels.npy` dataset as well as the `story_atlas.nii.gz` NIfTI image from which the parcels were derived (for later visualization).

```
# Load in parcel time series:
parcels = np.load("story_parcels.npy")

# Load in the Schaefer 400-parcel atlas:
import nibabel as nib

atlas = nib.load("story_atlas.nii.gz")
```

Our word embedding "model" is much wider than the number of samples, so we'll need to use regularization and out-of-sample prediction to mitigate overfitting. We'll use ridge regression to fit encoding models to predict the parcel time series from the word embeddings. First, set up an split-half outer cross-validator using `KFold` with `n_splits=2`; next, set up an inner cross-validator using `KFold` with `n_splits=5` to perform grid search for the `alpha` hyperparameter using 5-fold cross-validation within each training set of the otuer loop. Initialize your `RidgeCV` estimator with the inner cross-validator and the following grid of alphas: `alphas = [0.1,` `1.0, 10.0, 100.0, 1000.0, 10000.0]`. For each training and testing split of the other cross-validation loop, fit the ridge model on the training set of embeddings and parcel time series, and generate predicted parcel time series from the test embeddings. Compile these predicted parcel time series for model evaluation in the next step:

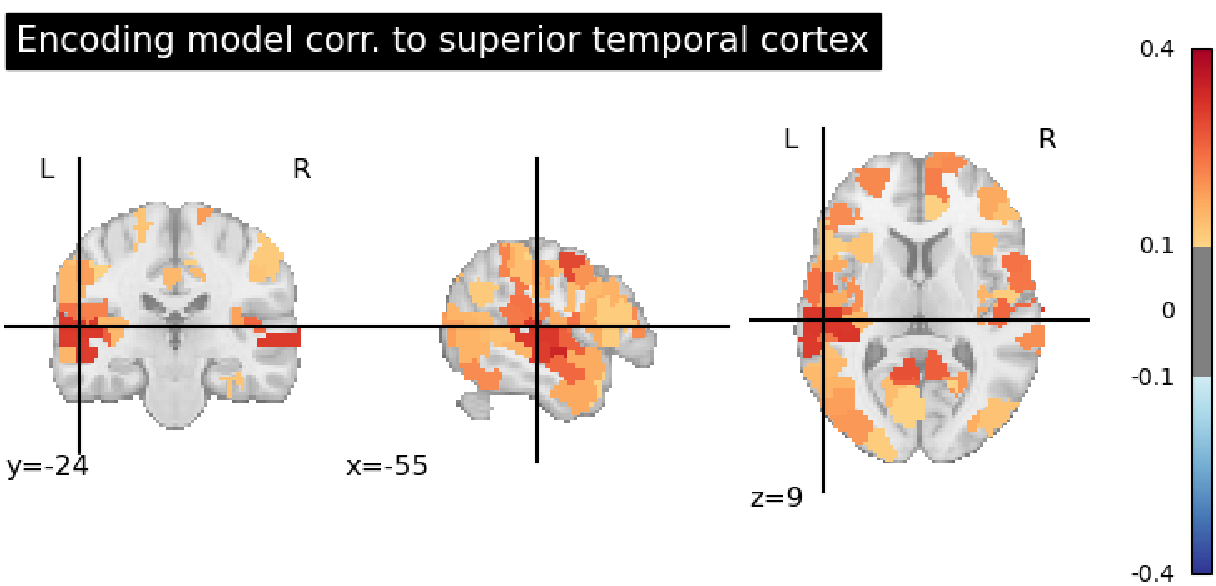
```
# Set up outer/inner cross-validators:
from sklearn.model_selection import KFold

outer_cv = KFold(n_splits=2)
inner_cv = KFold(n_splits=5)

# Initialize RidgeCV with alpha grid and inner CV:
from sklearn.linear_model import RidgeCV

alphas = [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
ridge = RidgeCV(alphas=alphas, cv=inner_cv)

# Loop through outer CV loop, fit model, generate predictions:
func_predicted = []
for train, test in outer_cv.split(parcels):

    # Fit ridge regression with grid search
    ridge.fit(embeddings[train],
              parcels[train])

    # Compute predicted response
    predicted = ridge.predict(embeddings[test])
    func_predicted.append(predicted)

# Restack first and second half predictions
func_predicted = np.vstack(func_predicted)
```

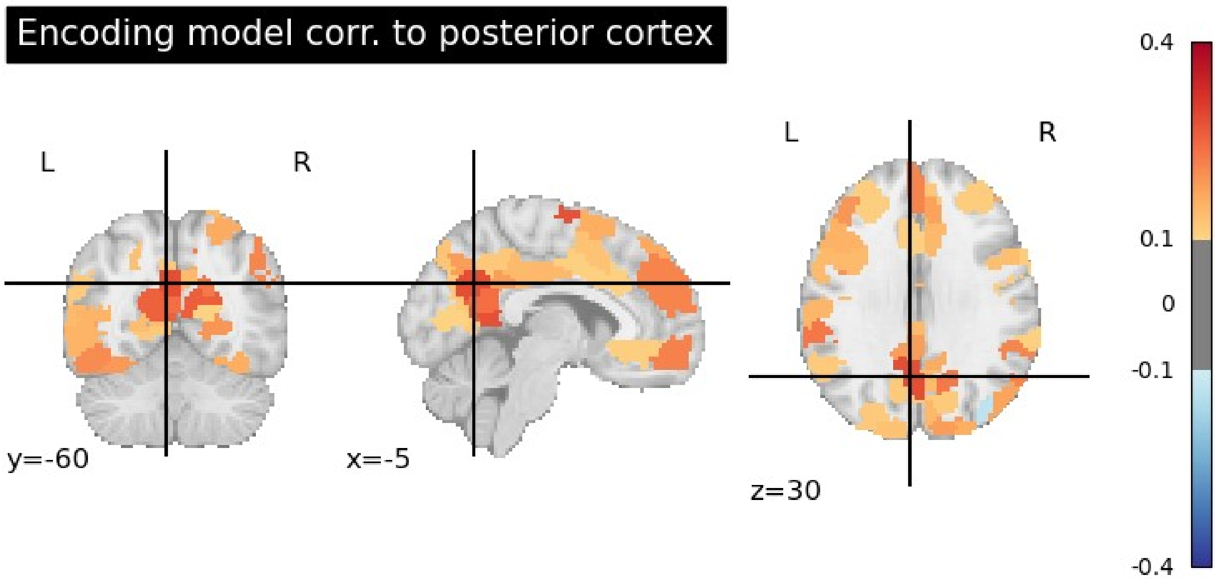
To evaluate our encoding model's predictions, correlate the predicted parcel time series with the actual parcel time series for each parcel.

```
# Compute correlation between predicted and actual responses:
from scipy.stats import pearsonr

r_parcels = []
for parcel in np.arange(parcels.shape[1]):
    r_parcels.append(pearsonr(parcels[:, parcel], func_predicted[:,
parcel])[0])
```

Finally, to visualize the performance of our semantic encoding model on the brain, we need to use the atlast NIfTI image to convert from parcels back to the original brain image. You can start by creating an empty brain image (i.e. zeros) the size of the atlas image. Next, loop through each parcel and insert the prediction scores (i.e. correlations between actual and predicted parcel time series) into all voxels where the atlas correponds to that parcel label. Convert this image to a NIfTI image and visualize with `plot_stat_map`; you may want to set a particular `vmax` and use a `threshold` to exclude voxels with poor prediction performance for the sake of visualization.

```
# Create an empty brain image and populate with parcelwise performance
values:
from nilearn.plotting import plot_stat_map

# Convert to NIfTI image for visualization with Nilearn:
r_img = np.zeros(atlas.get_fdata().shape)
for i, parcel in enumerate(np.unique(atlas.get_fdata())[1:]):
    r_img[atlas.get_fdata() == parcel] = r_parcels[i]

r_nii = nib.Nifti1Image(r_img, atlas.affine, atlas.header)

# Plot correlations to visualize superior temporal cortex
vmax = .4
threshold = .1
plot_stat_map(r_nii, cmap='RdYlBu_r', vmax=vmax, threshold=threshold,
              title='Encoding model corr. to superior temporal
cortex', cut_coords=(-55, -24, 9))

# Plot correlations to visualize posterior medial cortex
plot_stat_map(r_nii, cmap='RdYlBu_r', vmax=vmax, threshold=threshold,
              title='Encoding model corr. to posterior cortex',
cut_coords=(-5, -60, 30))

<nilearn.plotting.displays._slicers.OrthoSlicer at 0x12050ab50>
```




Encoding model corr. to posterior cortex
L
R
y=-60
x=-5
L
R
z=30
0.4
0.1
0
-0.1
-0.4