In [ ]:
```python
import numpy as np
from hw11_utils import ImageTagger
```

1. Tagging images In this exercice you will "tag" images with one of the possible values of the feature that you and your group proposed on Piazza. These images come from a real experiment that was done by a member of the Gallant lab here at UC Berkeley some years ago (here's a link to the paper). The images span many different semantic categories (such as human face, land mammal or water scene, for example). The original study used an encoding model (which we learned about in lecture 12) to find regions of visual cortex that are selective to the 19 semantic categories that the researchers used to "tag" each image. The goal of this exercise is to tag all of the images from this experiment using the feature that you have chosen. The next (and last!) homework of the semester will ask you to use these "tags", as well as the "tags" from the other students in class, to build a design matrix that you can use to fit an encoding model of the actual fMRI data from the original study.

In the first code cell below we've encoded 4 features along with all the possible values each feature can take (for example, the feature "evoked_emotion" can take one of 7 values like 'Anger' or 'Happiness'). Leave that cell alone, as these are the features that each group has either suggested, or that we've assigned to them since they didn't post anything to Piazza. The group assignments are as follows:

`outdoors` : Adelaide Chen, Dominic LeDuc, Nachiket Mehta

`evoked_emotion` : Riley McDanal, Eric Wimsatt, Apoorva Polisetty

`reward` : Tamara Gerbert, Wesley Thomas, Agnes Wiberg

`curves vs lines` : William Ryan, Hannah Liu, Jessica Singh, Amy Egan

We require that each group rate all of the images from the experiment. There are 1386 images in total. You may split these images up equally amongst all of the members of your group (so each of you rates 1/3 of the images), or if you each want to rate all the images then we will use the mean (or mode) rating, which will likely mean that we are more likely to find meaningful results in the brain data (why would that be?). In order to tag just a subset of the images you can specify the indices of the images you want to tag in the `my_range` name. If you just want to tag every 3rd image, you can use one of the pre-defined values for `my_range` defined below. There are 3 pre-defined values, and each group member should use a different value, which start 0, 1, or 2.

To start tagging the images, run all the cells below. In the output of the final cell containg simply `it` you should see a dropdown menu item, some buttons, and a single image to tag. First select which of the four above mentioned features you will be rating. To do so, you can either select the feature name from "Feature" dropdown menu located at the top of the cell's output, or by modifying the `my_feature_type` name below to say the name of the feature you are assigned to (this will simply default the dropdown menu to the feature name you store in the `my_feature_type` name).

Once you have selected you group's feature you are ready to start tagging. Simply select the appropriate tag for the current image by pressing the button with the text for the desired tag. This will cause the next image to be displayed. The blue progress bar indicates your progress. To the right of it you see how many images are remaining. Continue tagging images until it says "DONE" next to the blue progress bar (it will say "DONE" instead of how many images are remaining).

IMPORTANT: There is a save-button at the bottom. Use it regularly, or run the `it.save_tags()` command below from time to time. This saves your tags to disk, and prevents you from loosing and data you've tagged in the event of an error. We don't expect you to encounter any errors, but frequent saving is always a good idea, with any computer program! Once you have saved, you can close the browser or even restart the kernel of the notebook and the tags will be loaded from disk, so you won't lose your work.

ONCE FINISHED: When you are done, click on the jupyter icon to go to your root folder. Find and download the file `LH_tags.json` . Upload this file to bcourses.

You can also navigate through the image set using the following buttons:

The double-arrow buttons bring you to the next/previous untagged image.

The single-arrow buttons bring you to the next image among the ones you wanted to tag (`my_range`).

The slider gets you to whichever image you want. You can also just write the index you want directly.

Above the slider there is a progress bar indicating roughly how far you are with your task (as a proportion of the values in `my_range`).

The long bar in the "statu" tab shows you which images have been tagged and which ones still need tagging (green/red). Parts that are not within `my_range` are grayed out.

In [ ]:

```
tag_specs = dict()
tag_specs['outdoors'] = ['outdoors', 'indoors', 'under water', 'unclear', 'not a scene', '
tag_specs['evoked_emotion'] = ['Anger', 'Disgust', 'Fear', 'Happiness', 'Sadness', 'Surpri
tag_specs['reward'] = ['0', '1', '2', '3', '4', 'untagged']
tag_specs['curves_vs_lines'] = ['curved', 'mix', 'lines', 'untagged']
```

In [ ]:

```
# You can change this to the name of the feature your group is assigned,
# or simply select your feature from the dropdown menu created below (after running the be
my_feature_type = ('outdoors', 'reward', 'evoked_emotion')
```

In [ ]:

```
# tag all of the images
my_range = range(0, 1386, 1)

# tag every 3rd image starting with the first image
# my_range = range(0, 1386, 3)
# tag every 3rd image starting with the second image
# my_range = range(1, 1386, 3)
# tag every 3rd image starting with the third image
# my_range = range(2, 1386, 3)
```

In [ ]:

```
image_tagger = ImageTagger(tag_specs, my_feature=my_feature_type, ok=ok, tag_range=my_rang
```

In [ ]:

```
# This cell will display the ImageTagger where you will tag all the images
image_tagger
```

In [ ]:

```
# Just in case you forgot to save, evaluating this cell does it, too
image_tagger.save_tags()
```

If necessary, you can remove the tag file and start over by running # !rm /home/jovyan/LH_tags.json

In [ ]:
```
import numpy as np
import cortex
import nibabel
from nistats.hemodynamic_models import glover_hrf as create_hrf
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:
```
def load_nifti(filename, zscore=True, mask=None):
    img = nibabel.load(filename)
    data = img.get_data().T
    if mask is not None:
        data = data[:, mask]
    if zscore:
        data -= data.mean(0)
        data /= data.std(0) + 1e-8
    return data
```

2. Investigating overfitting In this exercise you will take a closer look at overfitting and how predicting on a held-out test set can detect and alleviate it.

Generally we call this *out-of-sample* evaluation: The data used to fit the model were not the data used to evaluate the model. This is in contrast to *in-sample* evaluation, where the error is computed on the same data as the model was fit on. We will see that in-sample evaluation is often "better" in terms of error than out-of-sample evaluation, but that it can lead to models that are not predictive at all. Even though the error is worse, it is often safer to evaluate models on a held-out set.

(a) Load the design matrix Load the motor labels from `"/data/cogneuro/fMRI/motor/motorloc_experimental_conditions.npy"` , remove the first 10 and the last 15 labels. These are additional rest periods before and after the scan. Store the result in `motor_labels` .

Store the unique names of the motor tasks from `motor_labels` into `unique_motor_labels` and make the stimulus design matrix from `motor_labels` . Call it `stimulus_design_full` .

Split the design matrix into training and test data sets by taking the top 3/5 of the design matrix (180 TRs) and store it in `stimulus_design_train` . Then take the bottom half of it (120 TRs) and store it in `stimulus_design_test` . Addtionally create a stimulus design matrix `stimulus_design_overlap` which takes TRs from `60` to `240` . This overlaps the train set by 2/3 and the test set by 1/3 and has the same size as `stimulus_design_train` . If it predicts better on the test set, then we know it is fitting particularities of the test set.

Create an `hrf` of time length 32, and tr=2 and use it to create 4 different response design matrices called `response_design_full` , `response_design_train` , `response_design_test` , and `response_design_overlap` .

In [ ]:
```
motor_labels = np.load("/data/cogneuro/fMRI/motor/motorloc_experimental_conditions.npy")[1
unique_motor_labels = np.unique(motor_labels)
stimulus_design_full = motor_labels.reshape(-1, 1) == unique_motor_labels
stimulus_design_train = stimulus_design_full[:180]
stimulus_design_test = stimulus_design_full[180:]
stimulus_design_overlap = stimulus_design_full[60:240]

hrf = create_hrf(tr=2, oversampling=1, time_length=32)

response_vectors_full = []
```

```
response_vectors_train = []
response_vectors_test = []
response_vectors_overlap = []

for i in range(len(unique_motor_labels)):
    response_vectors_full.append(np.convolve(stimulus_design_full[:, i], hrf)[:len(stimulu
    response_vectors_train.append(np.convolve(stimulus_design_train[:, i], hrf)[:len(stimu
    response_vectors_test.append(np.convolve(stimulus_design_test[:, i], hrf)[:len(stimulu
    response_vectors_overlap.append(np.convolve(stimulus_design_overlap[:, i], hrf)[:len(s


response_design_full = np.stack(response_vectors_full, axis=1)
response_design_train = np.stack(response_vectors_train, axis=1)
response_design_test = np.stack(response_vectors_test, axis=1)
response_design_overlap = np.stack(response_vectors_overlap, axis=1)
```

(b) Load the Data Using `load_nifti` , load and mask the motor localizer data in the filename `/data/cogneuro/fMRI/motor/s01_motorloc.nii.gz` and remove the first ten and the last fifteen measurements. Store the output in `motor_data_full` .

Extract the time series of voxels with indices `[34854, 37594, 36630, 25004, 12135, 0]` and call it `voxels_full` .

Then split `motor_data_full` into `motor_data_train` (first 180 TRs), `motor_data_test` (last 120 TRs) and `motor_data_overlap` (TRs 60 to 240).

Perform the same split of `voxels_full` into `voxels_train` , `voxels_test` and `voxels_overlap` .

In [ ]:

```
mask = cortex.db.get_mask('s01', 'catloc', 'cortical')
motor_data_full = load_nifti("/data/cogneuro/fMRI/motor/s01_motorloc.nii.gz", mask=mask)[1
voxels_full = motor_data_full[:, [34854, 37594, 36630, 25004, 12135, 0]]

motor_data_train = motor_data_full[:180]
motor_data_test = motor_data_full[-120:]
motor_data_overlap = motor_data_full[60:240]

voxels_train = voxels_full[:180]
voxels_test = voxels_full[180:]
voxels_overlap = voxels_full[60:240]
```

(c) Fit Models and Predict Now you will fit three linear models. The first one, `lr_cv` will train on the train set. The second one, `lr_full` will train on the full design, and the last one, `lr_test` will train on the test set.

In a predictive modeling context, you need to split your data perfectly - any overlap will make the error go down, and we won't know whether it is overfitting or not. You will be able to see this when comparing the fit of `response_design_overlap` to the fit of `response_design_train` .

Create the `LinearRegression` estimator `lr_cv` and fit it to `response_design_train` and `voxels_train` .

Create the `LinearRegression` estimator `lr_full` and fit it to `response_design_full` and `voxels_full` .

Create the `LinearRegression` estimator `lr_test` and fit it to `response_design_test` and `voxels_test` .

Create the `LinearRegression` estimator `lr_overlap` and fit it to `response_design_overlap` and `voxels_overlap` .

Use all four to predict `response_design_test` and call the predictions `pred_train_test`, `pred_full_test`, `pred_test_test`, `pred_overlap_test` respectively.

In [ ]:

```
lr_cv = LinearRegression().fit(response_design_train, voxels_train)
lr_full = LinearRegression().fit(response_design_full, voxels_full)
lr_test = LinearRegression().fit(response_design_test, voxels_test)
lr_overlap = LinearRegression().fit(response_design_overlap, voxels_overlap)

pred_train_test = lr_cv.predict(response_design_test)
pred_full_test = lr_full.predict(response_design_test)
pred_test_test = lr_test.predict(response_design_test)
pred_overlap_test = lr_overlap.predict(response_design_test)
```

(d) Plot Results Make a figure named `fig_predictions` of size (24, 24). In it, make 6 subplots (each subplot shoudl be a row that represents each voxel from `voxels_test`) and in each subplot make one time series plot containing the current voxel's time course from `voxels_test` as well as the three predictions for that voxel from the 3 models you fit in part #c.

Make a legend and label your plots.

Compute the sum of squared errors for each prediction of each voxel and list the four SSEs in the title of each subplot.

In [ ]:

```
fig_predictions = plt.figure(figsize=(24, 24))
for i in range(voxels_full.shape[1]):
    plt.subplot(6, 1, i + 1)
    plt.plot(voxels_test[:, i], label="voxel {}".format(i))
    plt.plot(pred_train_test[:, i], label="train")
    plt.plot(pred_full_test[:, i], label="full")
    plt.plot(pred_test_test[:, i], label="test")
    plt.plot(pred_overlap_test[:, i], label="overlap")


    sse_cv = np.sum((pred_train_test[:, i] - voxels_test[:, i]) ** 2)
    sse_full = np.sum((pred_full_test[:, i] - voxels_test[:, i]) ** 2)
    sse_test = np.sum((pred_test_test[:, i] - voxels_test[:, i]) ** 2)
    sse_overlap = np.sum((pred_overlap_test[:, i] - voxels_test[:, i]) ** 2)

    sse_0 = np.sum((voxels_test[:, i] - voxels_test[:, i].mean(0)) ** 2)
    plt.title("voxel {} sse: 0: {:0.2f} cv: {:0.2f} ol {:0.2f} full: {:0.2f} test: {:0.2f}
        i, sse_0, sse_cv, sse_overlap, sse_full, sse_test))
    plt.legend()
```

3. Even more overfitting In this exercise you will add more and more noise columns to the response designs and find that while training error decreases, testing error increases.

(a) Add noise to the motor localizer data

Create a name called `n_noise_columns` and set it's value to 10.

Make two random arrays called `noise_train` and `noise_test` , the first of which is of size `(response_design_train.shape[0], n_noise_columns)` and the second of which is of size `(response_design_test.shape[0], n_noise_columns)` .

Create a new name called `noisy_design_train` by concatenating `response_design_train` with `noise_train` **horizontally** (along `axis=1`) using `np.concatenate` . Do the same for the test data and call the resulting name `noisy_design_test` .

In [ ]:
```
n_noise_columns = 10
noise_train = np.random.randn(response_design_train.shape[0], n_noise_columns)
noise_test = np.random.randn(response_design_test.shape[0], n_noise_columns)

noisy_design_train = np.concatenate([response_design_train, noise_train], axis=1)
noisy_design_test = np.concatenate([response_design_test, noise_test], axis=1)
```

(b) Calculate SSE

Fit a `LinearRegression` model using `noisy_design_train` as the independent and `voxels_train` as the dependent data.

Use this model to predict the training data, using `noisy_design_train` , and call the predictions `train_pred` .

Also predict the test data, using `noisy_design_test` and call the predictions `test_pred` .

Compute the sum of squared errors between `train_pred` and `voxels_train` and `test_pred` and `voxels_test` . Call them `sse_train` and `sse_test` respectively.

Print the SSE values.

In [ ]:
```
lr = LinearRegression().fit(noisy_design_train, voxels_train)
train_pred = lr.predict(noisy_design_train)
test_pred = lr.predict(noisy_design_test)
sse_train = np.sum((train_pred - voxels_train) ** 2) # several voxels summed up. One can a
sse_test = np.sum((test_pred - voxels_test) ** 2)

print(sse_train, sse_test)
```

(c) Put it into a function

Using what you did in **(a)** and **(b)**, create a function `noisy_fit` which takes as an argument `n_noise_columns` , creates the noisy design matrices, performs the regressions and predictions and outputs the sum of squared errors on train an test set.

In [ ]:
```
def noisy_fit(n_noise_columns):
    noise_train = np.random.randn(response_design_train.shape[0], n_noise_columns)
    noise_test = np.random.randn(response_design_test.shape[0], n_noise_columns)

    noisy_design_train = np.concatenate([response_design_train, noise_train], axis=1)
    noisy_design_test = np.concatenate([response_design_test, noise_test], axis=1)

    lr = LinearRegression().fit(noisy_design_train, voxels_train)
    train_pred = lr.predict(noisy_design_train)
    test_pred = lr.predict(noisy_design_test)
    sse_train = np.sum((train_pred - voxels_train) ** 2) # or axis=0
    sse_test = np.sum((test_pred - voxels_test) ** 2) # or axis=0

    return sse_train, sse_test
```

(d) Calculate SSE with variable numbers of Noise Variables Calculate the SSE of a linear model on the localizer data that adds incrementally more noise independent variables to see the effect adding additional noise independent variables has on the SSE of the training and test data sets. To do this, use the `noisy_fit` function in a `for` loop that iterates over i ranging from 0 to 150 and store the output of the function into two lists, one that stores the training SSE values called `all_sse_train` and one that stores the test SSE values called `all_sse_test` .

In [ ]:

```
all_sse_train = []
all_sse_test = []

for i in range(150):
    sse_train, sse_test = noisy_fit(i)
    all_sse_train.append(sse_train)
    all_sse_test.append(sse_test)
```