Imagined Emotion

A Brain-Computer Interface Project

DataSet

- Imagined Emotion
  - Link: http://headit.ucsd.edu/studies/3316f70e-35ff-11e3-a2a9-0050563f2612
  - Experiment: The experiment basically make subjects listen to voice recordings that suggest an emotional feeling and ask subjects to imagine an emotional scenario or to recall an experience in which they have felt that emotion before.
  - Potential use: Show how one can classify emotion states from acquired electroencephalographic data, and its possible relevance to finding new methods of reducing anxiety, stress, depression, and anger management.
  - Related publication:
    - Onton J and Makeig S (2009). High-frequency broadband modulations of electroencephalographic spectra. Front. Hum. Neurosci. 3,:61.
    - Onton J., Mackeig S., Independent modulators of regional EEG alpha sub-band power during a working memory task. Poster session presented to Cognitive Neuroscience Society; 2009 March; San Francisco, CA.

Related Works

Onton J and Makeig S (2009). High-frequency broadband modulations of electroencephalographic spectra. Front. Hum. Neurosci. 3,:61.

- Collected our dataset
- Details how the data was collected
  - 250 Scalp electrodes, 4 infra-oscular, and 2 ECG at 256Hz
- Ran ICA decomposition
- Suggests that high frequency brain waves, i.e. beta, gamma, and high-gamma, can be used to reliably distinguish brain activity from ocular motions and scalp movements
- These brain waves were related to valence of imagined emotion

This suggests that it is possible to achieve classification based on the valence of each emotion

Load the dataset

The dataset was in BDF extention. (Biosemi file)

Goal: convert it into a readable format.

Step 1:

Write codes in Jupyter Notebook Python 3.7 to convert the bdf file into pandas dataframe.

In [74]:
```
def bdf_to_df(file_path):
    file_name = os.path.join('data', file_path)
    f = pyedflib.EdfReader(file_name)
    n = f.signals_in_file
    signal_labels = f.getSignalLabels()
    sigbufs = np.zeros((n, f.getNSamples()[0]))
    for i in np.arange(n):
        sigbufs[i, :] = f.readSignal(i)
    rec = pd.DataFrame(sigbufs).T
    rec.columns = signal_labels
    return rec
```

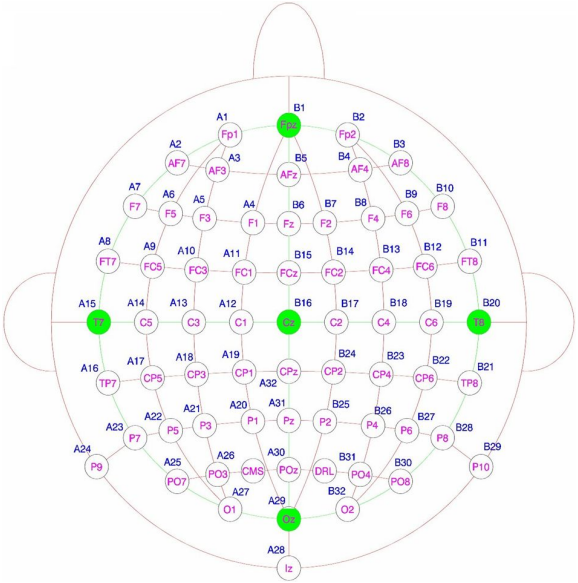
Data Cleaning: Explore the dataset

There are over 20+ datasets, each record the EEG activity of one patient. We will clean the dataset one by one. For the first recording, there are 1155328 unique rows, and 265 columns.

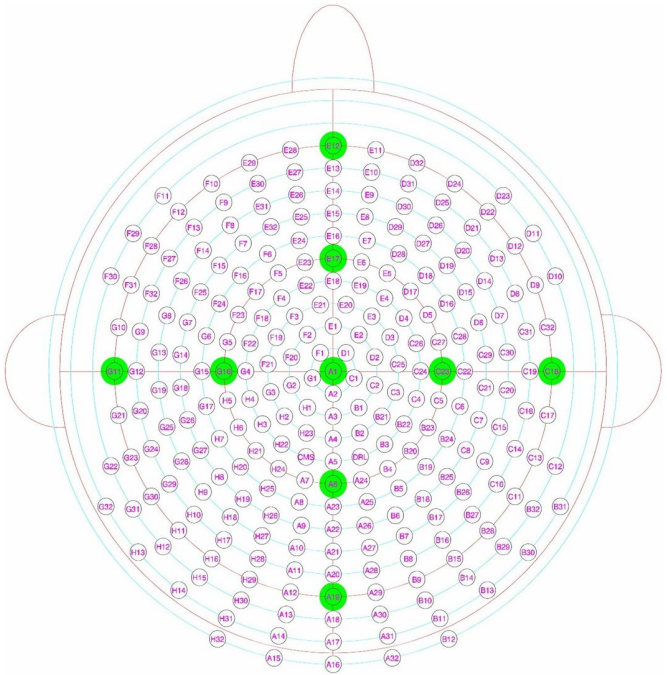
Problem: we have too much data, how can we extract useful information?

	1A1	1A2	1A3	1A4	1A5	1A6	1A7	1A8
0	-18876.730746	-18592.168772	-16102.454623	-15834.736367	-12413.086440	-9622.435346	-3279.728316	-15042.894080
1	-18882.855735	-18598.293761	-16107.892113	-15841.642605	-12419.711427	-9630.841581	-3288.322050	-15048.550320
2	-18881.918237	-18599.512509	-16108.860861	-15843.142602	-12420.930175	-9633.029077	-3290.103297	-15048.050321
3	-18873.762002	-18591.293774	-16098.454630	-15833.830119	-12411.117693	-9624.247843	-3280.290815	-15039.456587
4	-18872.543254	-18589.731277	-16096.360884	-15829.517627	-12408.898947	-9618.935353	-3276.759571	-15038.737838
5	-18880.543239	-18599.012510	-16103.423371	-15836.205115	-12416.930183	-9625.966590	-3283.353309	-15047.706572

64 channels vs 256 channels:



The 10 20 system covered in class



The 256 electrodes system used in our dataset

Column transformation

We will find the most useful 64 channels/columns to run our ML, because the original 256 columns are unnecessary.

Attempt: find the corresponding channels from the channel_location.elp file

But It did not give any useful information

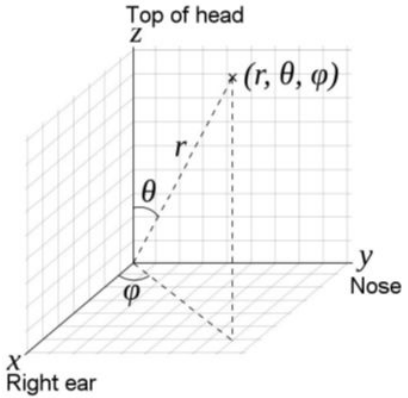
The function below `load_elp` tries to find all the labels from the elp file.

```
#this function tries to extract info from the elp file, but i dont really know t
#these coordinate to 10 20 system, so i will seek for alternative method
def load_elp(file_path):
    file_name = os.path.join('data', file_path)
    #open the elp file as a python string
    s = open(file_name, 'r').read()
    start_idx = s.index('//Sensor type')
    #split the Long string to find each sensor
    splitted = s[start_idx:].split('//Sensor type')
    splitted = [s for s in splitted if '#' in s]
    lst = []
    for line in splitted:
        sublst = []
        num = line[line.index('#'): line.index('#') + 5]
        num = int(re.findall(r'\b\d+\b', num)[0])
        sublst.append(num)
        typ = line[line.index('Sensor') - 8 : line.index('Sensor') - 3]
        sublst.append(typ)
        name = line[line.index('n%N'):]
        sublst.append(name)
        lst.append(sublst)
    return lst
```

```
%N      A16
0.0434  0.0687  0.0171
//Sensor type
%S      400
//Sensor name and data for sensor # 19
%N      A17
0.0495  0.0723  0.0437
//Sensor type
%S      400
//Sensor name and data for sensor # 20
%N      A18
0.0558  0.0710  0.0594
//Sensor type
%S      400
```

Column transformation

We looked up the location spatial coordinate, and we find the channel in 256 system with the closet euclidean distance to every channel in the 64 electrodes system. This is done through writing codes.



The coordinates

Electrode	θ (Inclination)	φ (Azimuth)
Fp1	-92	-72
AF7	-92	-54
AF3	-74	-65
F1	-50	-68
F3	-60	-51
F5	-75	-41
F7	-92	-36

Some coordinate values

Columns remaining: 64

In [88]:
```
rec1_64 = rec1.filter(['1' + i for i in bestcol64] + ['Status'])
rec1_64 = rec1_64.reindex(sorted(rec1_64.columns), axis=1)
```

In [89]:
```
rec1_64
```

Out[89]:

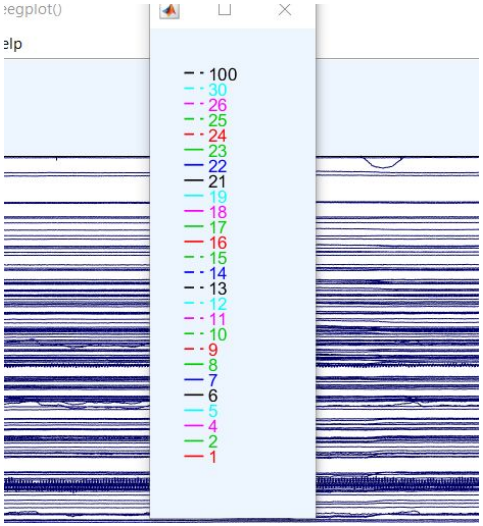
	1A1	1A16	1A19	1A21	1A24	1A3
0	-18876.730746	-16806.859571	-3981.008270	-5758.473736	-8187.469248	-16102.454623
1	-18882.855735	-16812.609560	-3986.383260	-5766.004972	-8196.875480	-16107.892113
2	-18881.918237	-16813.140809	-3988.039507	-5766.161222	-8198.156728	-16108.860861
3	-18873.762002	-16804.172076	-3978.570775	-5756.411240	-8187.687997	-16098.454630
4	-18872.543254	-16803.297078	-3977.320777	-5753.411245	-8185.156752	-16096.360884
5	-18880.543239	-16812.172061	-3983.695765	-5759.161234	-8189.844243	-16103.423371
6	-18882.824485	-16814.515807	-3985.789511	-5761.754980	-8191.875489	-16108.954611
7	-18880.418240	-16811.547062	-3980.852020	-5757.004988	-8192.250489	-16105.829616
8	-18874.043251	-16807.453320	-3974.664532	-5751.098749	-8187.656747	-16095.954635

Data cleaning: status code

Problem: we got a status column with values such as 3407872.0, 3407873.0, 3407874.0, 3407876.0, 3407877.0, 3407878.0, 3407879.0, while the experience instruction give code like 0, 1, 2, 3, to mark different events(emotions) and the status code vary for different data frames.

Solution: use `Matlab EEGLAB` to open the bdf file and find the most frequent even code, and map it to dataframe.

Explore the data using EEGLAB

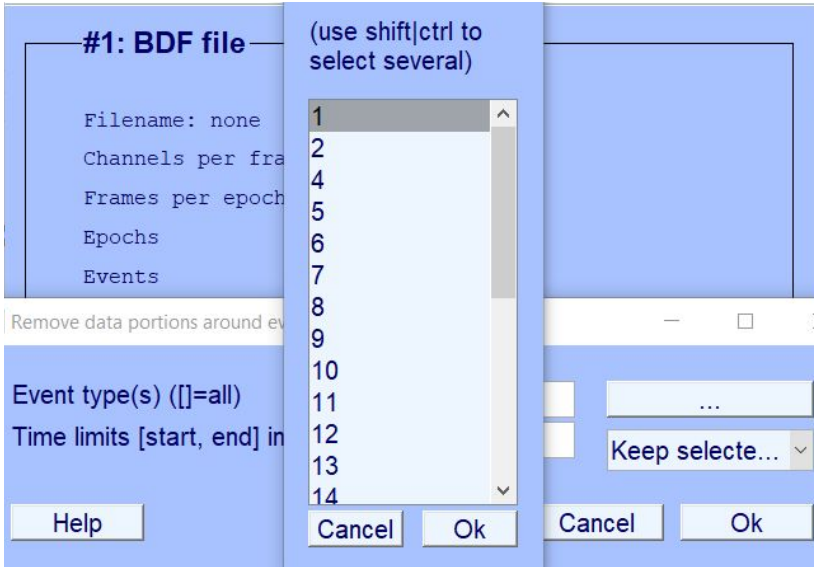


Correct event status value

Explore the data using EEGLAB

Method: go to edit -> select data using event -> remove data portion around events -> select each event code

By going over each event code one by one we can check the difference in channels per epoch and sample rate, therefore determine which value represents the correct event status code.



Data ready for use

We clean the recording data 1, 2, 3, 4, 5, 8, 9 because these are the patients' recording that we are able to translate the event code to the correct signal. We remove the most frequent signal label, which only produce noise and took too much storage.

Binarize the prediction

1	Event Code	Label	Description	Tag
2	0	button press	ends each emotion scenario and prompts track 2...	Response/Button press
3	1	introduction track	sound clip explaining initial instructions and...	Stimulus/Instruction, Stimulus/Auditory/Langua...
4	2	pre-baseline	sound clip instructing subject to sit still fo...	Stimulus/Task/Task Rest Start, Stimulus/Instru...
5	3	instruction 1a	sound clip that asks subject to press the butt...	Stimulus/Instruction, Stimulus/Auditory/Langua...
6	4	instruction 1b	sound clip that asks for subject to press the ...	Stimulus/Instruction, Stimulus/Auditory/Langua...
7	5	instruction 2	sound clip explaining the complete experiment ...	Stimulus/Instruction, Stimulus/Auditory/Langua...
8	6	relax	sound clip that generally guides relaxation an...	Stimulus/Instruction/Imagine, Stimulus/Auditor...
9	7	instruction 3	sound clip introduces the emotion episodes	Stimulus/Instruction, Stimulus/Auditory/Langua...
10	8	'prepare'	sound clip plays before each emotion episode a...	Stimulus/Instruction, Priming/Emotional
11	9	awe	a sound clip that describes the postive emotio...	Stimulus/Onset, Stimulus/Auditory/Language/Sen...
12	10	frustration	a sound clip that describes the negative emoti...	Stimulus/Onset, Stimulus/Auditory/Language/Sen...
13	11	joy	a sound clip that describes the postive emotio...	Stimulus/Onset, Stimulus/Auditory/Language/Sen...
14	12	anger	a sound clip that describes the negative emoti...	Stimulus/Onset, Stimulus/Auditory/Language/Sen...
15	13	happy	a sound clip that describes the postive emotio...	Stimulus/Onset, Stimulus/Auditory/Language/Sen...

Predict positive/negative emotion

We used decision tree classifier to predict the positive (1) emotion or negative (0) emotion from the concatenated 7 dataframes we made. Our best prediction score is 1.0 on the training data, and 0.966 on the testing dataset. (testing size = 25%)

To prevent overfitting, we also tried the prediction on not all of the dataset, the result was similar.

How fair is our binarized classifier?

Sensitivity is what is the proportion of actual positive emotions that are correct identified? Precision is when we label an event as positive emotion, what is the proportion that we actually get it correct?

```
In [130]: #proportion of actual positive emotions that are correct identified
          metrics.recall_score(y_test, preds)

Out[130]: 0.9215686274509803

In [132]: #proportion of actual negative emotions that are correct identified
          metrics.recall_score(y_test, preds, pos_label=0)

Out[132]: 0.8901098901098901

In [134]: #proportion of predicted positive emotions that are actually positive emotions
          metrics.precision_score(y_test, preds)

Out[134]: 0.8245614035087719

In [135]: #proportion of predicted positive emotions that are actually negative emotions
          1 - metrics.precision_score(y_test, preds)

Out[135]: 0.17543859649122806
```

Next: More Machine Learning

If we binarize the prediction of emotion on positive and negative, the results looks good. What if we want to predict all the emotions it includes?

For the following analyses, we will be using testing classification accuracy as our performance metric.

Setting up the dataframe

Before starting machine learning analysis, we must prepare our dataframe for the classification task.

```
rec1 = pd.read_csv('rec1.csv')
rec2 = pd.read_csv('rec2.csv')
rec3 = pd.read_csv('rec3.csv')
rec4 = pd.read_csv('rec4.csv')
rec5 = pd.read_csv('rec5.csv')
rec8 = pd.read_csv('rec8.csv')
rec9 = pd.read_csv('rec9.csv')
print(rec1.shape, rec2.shape, rec3.shape, rec4.shape, rec5.shape, rec8.shape, rec9.shape)

# Concatenate the 4 participants into one dataframe
df = pd.concat([rec1, rec2, rec3, rec4, rec5, rec8, rec9], ignore_index=True)
print(df.shape)
```

Because we are attempting to classify categorical emotions, it is important to have data from multiple subjects so that our classification has better external validity.

In our case, we are combining data from 7 different subjects, giving us 3282 observations (rows), 65 features (columns), and 27 targets (emotions) to work with.

Training and Testing Split

In accordance to typical machine learning practices, we must reserve some of our dataset for training, and the rest for testing and validation.

```
# Separate dataframe into predictors and targets
X = df[df.columns[:-1]]
y = df[df.columns[-1]]

# Split predictors and targets into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

In our case, we will randomly shuffle 80% of the data for training our classifiers, and 20% for testing and validating the performance of our models.

This will also be helpful when we use cross validation to select stronger models later.

Starting Simple: Linear Models

When performing an analysis using machine learning, it is good practice to start by attempting a simpler model, i.e. a linear model, before moving onto more advanced algorithms and networks.

Starting with simpler models allows us to examine the nature of the dataset quickly and lead to more interpretable results.

In our case, we will attempt to build a model around three linear classifiers first: Linear Discriminant Analysis, Logistic Regression, and Linear Support Vector Machines.

LDA: Linear Discriminant Analysis

LDA is a linear, multiclass classifier that uses Bayesian probability to train a quick and simple model.

It is important that we use 5 fold cross validation to find the best 80% of data to train our classifier on such that it yields the best predictive power.

```
# LDA Classifier
clf_lda = LinearDiscriminantAnalysis()

# 5-fold cross validation
kf = KFold(n_splits = 5, shuffle=True, random_state=0)

train_scores = []
test_scores = []

# Obtain and iterate over train and test indices
for train_index, test_index in kf.split(X):

    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train LDA classifier
    clf_lda.fit(X_train, y_train)
```

LDA: Results

We obtained a train accuracy of 76.79% and a testing accuracy of 73.37%.

While these results are decent for 27 class classification, these scores could be held back by some of LDA's limitations.

```
# Obtain training and testing accuracy
train_acc_lda = np.mean(train_scores)
test_acc_lda = np.mean(test_scores)

print('Mean training score: ' + str(train_acc_lda))
print('Mean testing score: ' + str(test_acc_lda))
```

```
Mean training score: 0.7697289087150473
Mean testing score: 0.7336957530534209
```

LDA attempts to maximize the variance between groups, but also suffers by assuming that the predictors are normally distributed.

Our next classifier, Logistic Regression, does not make this assumption.

Logistic Regression

Like LDA, Logistic Regression is also a linear classifier, but does not assume normality like LDA does.

```
# Logistic Regression Classifier
clf_log = LogisticRegression(random_state=0, solver='saga', multi_class='multinomial', max_iter=1000)

# Parameters to tune
params = {
    'C': [10e-5, 10e-4, 10e-3, 10e-2, 10e-1]
}

# Grid Search for best parameter value
grid_log = GridSearchCV(clf_log, params, cv=5, n_jobs=-1, return_train_score=True, iid=False)
```

Logistic regression is not typically used for multiclass classification, but different solvers can be used for multiclass problems. We will be using the “saga” solver for its multiclass capabilities and regularization.

In order to potentially increase the performance of the classifier, we also tuned a L2 regularization parameter “C” using a grid search.

Logistic Regression: Results

Unfortunately, the results obtained turned out to be very similar to those for LDA.

We achieved a training accuracy of 71.62% and a testing accuracy of 73.51%.

```
# Obtain accuracy for training and testing
train_acc_log = result_log.score(X_train, y_train)
test_acc_log = result_log.score(X_test, y_test)

print('Best parameter value is: ' + str(result_log.best_params_))
print('Training accuracy: ' + str(train_acc_log))
print('Testing accuracy: ' + str(test_acc_log))
```

```
Best parameter value is: {'C': 0.0001}
Training accuracy: 0.7161904761904762
Testing accuracy: 0.7351598173515982
```

It is possible that there is a correlation between the EEG signals that is causing the Logistic Regression to fail to converge.

Perhaps it is time to move onto a more advanced algorithm...

Support Vector Machine (SVM)

SVM is a supervised machine learning model that is typically used for two-group classification problems. However in our project, we used SVM to train multiclasses yet the result is reasonably good.

During training, the whole dataset is partitioned into testing set and training set. The training set is fed into the SVM algorithm and the testing set is used for validation. The ratio of testing versus training is 1:3.

```
# partition
testing_data = data[:int(len(data)/4)]
training_data = data[int(len(data)/4):]

# partition
testing_label = label[:int(len(data)/4)]
training_label = label[int(len(label)/4):]
```

Output:

```
number of unique labels 27
number of data 3282
length of testing 820
length of training 2462
```

Model formula

```
clf = svm.SVC(kernel='linear', class_weight='balanced', C=1.0, random_state=0,decision_function_shape='ovo')
```

The SVM model we chose uses the following formula (**linear** kernel) to predict the class of a new input:

f(x) = B(0) + sum(ai * (x,xi))

In which x is the input and xi is the support vector.

The class weight is balanced, meaning the class weights will be given by:

```
n_samples / (n_classes * np.bincount(y))
```

The decision function shape is defined as ovo which means the algorithm automatically transform the result of one against one classifiers to a decision function of shape (n_samples, n_classes).

SVM result

```
clf.fit(training_data, training_label)
tmp = clf.predict(testing_data)
tmp = np.split(tmp,len(tmp),0)
result = []
for i in range(len(tmp)):
    result.append(tmp[i][0])

#print(testing_label)
#print(result)

acc = 0
for i in range(len(testing_label)):
    if testing_label[i] == result[i]:
        acc+=1

print('number of correct predction:' + str(acc))
print('accuracy '+str(acc/len(testing_label)))
```

```
number of correct predction:793
accuracy 0.9670731707317073
```

The result is stored and compared with the actual labels. If the two match accurate count is incremented by 1, otherwise it remains the same. Finally the accuracy count is divided by the total number of labels in the testing set to obtain the percent accuracy.

Nonlinear Machine Learning Approaches

While performance using Linear Support Vector machines was excellent, we can attempt to use more advanced nonlinear approaches to eliminate the effects of bias.

Random Forests

Unlike the previous classifiers, Random Forest is an ensemble classifier that uses multiple decision tree with limited feature sets to overcome overfitting.

```
# Random Forest Classifier
clf_rf = RandomForestClassifier(random_state=0)

# Hyperparameters for RF classifier
params = {
    'n_estimators': [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
    'max_depth': [10, 20, 30, 40, 50, 60, 70, 80 ,90, 100],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5]
}

# Randomized search to find optimal hyperparameters
rand_rf = RandomizedSearchCV(clf_rf, params, n_jobs=-1, iid=False, cv=5, return_train_score=True)
```

In addition, we will also be using a randomized search to tune the hyperparameters of our RF classifier.

Random Forest: Results

```
# Obtained optimal hyperparameter values
print('Best parameter values are: ' + str(result_rf.best_params_))

# Calculate accuracy of optimized RF model
train_acc_rf = result_rf.score(X_train, y_train)
test_acc_rf = result_rf.score(X_test, y_test)

print('Train Score: ' + str(train_acc_rf))
print('Test Score: ' + str(test_acc_rf))
```

```
Best parameter values are: {'n_estimators': 400, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_depth': 80}
Train Score: 0.9996191926884996
Test Score: 0.9679878048780488
```

As expected, Random Forests achieves scores better than those of LDA and Logistic Regression, and maintains an test score on par with those of our Linear SVM.

Discussion

- Accuracy
  - SVM and Random Forest outperforms LDA and Logistic Regression
    - Multi-Label Nature of the data
- Limitations
  - Data corruption
  - Data outdatedness
  - Data collection
  - Hardware constraints
    - Not enough processing power for Deep CNN algorithms
- Applications
  - Medical field
  - Entertainment

Speakers:

Jeffrey Feng, Leyi Shang, Justin Nguyen, Waylon Chang

BGM from Chenyu Hua “I Really Want to Love This World”