Gerald Pho and Mahmood Mohammadi Shad

1) Exploratory Data Analysis

Part a. Insight into given dataset

We started by looking carefully at the data. It's a clearly imbalanced dataset with two classes, 84% of them with output=1. Training data come from 11 different subjects, and while the test set contains some data from these subjects, the majority (54%) of the data is from new subjects E and J.

The data contains **222** numeric features each for x, y, and z, plus categorical features of **subject**, **state**, and **phase**. We found that significant subject-subject variability, with one subject (K) always in class 1, and others reliably in class 1 depending on the phase.

Part b. Over-Sampling of imbalanced dataset (Second Approach)

In the second approach, the imbalanced dataset is balanced using the Modified Synthetic Minority Oversampling Technique (MSMOTE). The minority class is classified into three different groups called safe, border, and latent noise samples. This classification is carried out by using the distance between minority class samples and training dataset. The safe samples basically improve the performance of the classifier while the latent noise ones reduce this performance. Any samples from minority class that does not belong to safe or noise goes under border class. This technique is applied using **imblearn** package in Python.

2) Feature cleaning, extraction, and engineering

We tried the following approaches for feature cleaning.

a) *Separated binary and numeric features* = Some features showed only a zero or one, so we assumed those were binary.

b) *Removed low variance features* = Many features had very low variance and so we dropped all features that showed variation on less than 0.5% of the data points.

c) *Renormalized numeric features by mean and standard deviation* = We did not normalize binary features.

d) *Removed features with strong collinearity* = Many features were strongly correlated with one another. We removed any with >0.95 Pearson correlation.

e) *Converted categorical variables to one-hot encoding* = We considered subject, state, and phase as categorical. We experimented with leaving subject in (to leverage within-subject consistency) versus out (to help generalize across subjects).

We also experimented with a few approaches for feature engineering.

f) *Engineered deterministic interaction features* = Some combinations of categorical features were highly deterministic – for example Subject A always was in class 1 when in phase 1. We generated new features based on the interaction of those two features.

g) *Engineered combined X-Y-Z features using LDA* = Although we don't know the meaning of the features, we guessed that X, Y, and Z corresponded to different axes for the same sensor. In

this case, it could be meaningful to work with a combined feature for each sensor. We chose to use LDA to reduce the dimensionality of each set of three X-Y-Z features to a single feature that maximally predicted the output.

h) *Prioritize predictive features (as assessed with ROC score) when removing collinear features* = rather than blindly removing collinear features, we sorted features by ROC score before elimination. We can also use the ROC score to rank features and eliminate features with low predictive value in case of overfitting.

i) In the second approach that uses over-sampling and PCA transform, we considered PCA transform with different number of components. Different number of components were tested, and 200 components were chosen as the final setting.

3) Train/validation split, cross-validation, and evaluation metrics

Train/validation splits = To assess the performance of our classifiers, we chose to try two different ways of splitting the data. For validation set #1, we did a random split of 75% train / 25% validation set. For validation set #2, we held out all data from two subjects ('A' and 'M' – which showed strong similarity to test subjects 'E' and J') and supplemented with random data points from other subjects until we reached 25% of the data. Validation set #1 therefore gives us an estimate of within-subject performance, and validation set #2 gives us an estimate of across-subject performance. We consistently found that performance on Validation set #1 overestimated performance on the online Kaggle set, whereas Validation set #2 underestimated performance. It may be most instructive therefore to consider the average of the two performances.

Cross-validation = We also considered different approaches for cross-validation. A simple stratified K-fold cross-validation does not take into account differences across subjects. We therefore considered multiple group-related splits, including GroupKFold and LeavePGroupsOut. Stratified K-Fold consistently overestimated performance, whereas LeavePGroupsOut underestimated performance. This tells us that there is a lot of predictive information within subjects that doesn't generalize well across subjects.

Evaluation metric = We chose to use ROC score (AUC) as the evaluation metric, as that matches the online Kaggle scoring system, and works well for this imbalanced setting.

In all cases, we used cross-validation and our validation set to optimize hyperparameters, but then re-trained models on the whole dataset (training + validation) to maximize test performance on the Kaggle set.

4) Handling class imbalance

Class imbalance is severe in this problem and will affect the performance of most classifiers. We therefore tried using Synthetic Minority Oversampling Technique (SMOTE) – the Nominal-Continuous (SMOTE-NC) version, which handles combinations of continuous and categorical features. I tried this once but it did not seem to improve performance of XGBoost, so I left it out.

In the second approach reported in the Jupyter notebook, the over-sampling is used with PCA transformation which yields a better result on both test dataset with and without subject holding.

5) Exploring different algorithms

Sklearn makes it easy to explore a large swath of different algorithms. We did a breadth-first search to identify promising candidates for our problem:

Algorithm	Valid1	Valid2	Notes
First Approach			
LDA	0.785	0.540	Does fairly well for such a simple algorithm!
QDA	0.673	0.550	Overfits terribly
Random Forests	0.814	0.594	Does fairly well
XGBoost	0.850	0.675	Using Rashmi's initial parameters
Second Approach			
Support Vector Machine (SVM)	0.772	0.615	Radial based function (rbf) kernel
SVM with PCA	0.765	0.616	SVM with PCA transformation (200 comps.)
Nu-Support Vector Classification (NuSVC)	0.749	0.602	Nu=0.1, same rbf kernel
NuSVC with PCA	0.746	0.604	NuSVC with PCA transformation
Gaussian Process	0.620	0.532	Maximum number of iterations: 10000
Gaussian Process with PCA	0.616	0.536	Gaussian process on PCA space, same no. iter.
AdaBoost Classifier	0.750	0.568	Adaptive boosting with 1000 estimators
AdaBoost Classifier with PCA	0.687	0.593	AdaBoost on PCA space and 1000 estimators
Naïve Bayes Classifier	0.673	0.582	Naïve Bayes with no priors
Naïve Bayes Classifier with PCA	0.582	0.569	Naïve Bayes on PCA scpace with no priors
KN Classifier	0.686	0.591	50 neighbors, uniform weight
KN Classifier with PCA	0.686	0.592	50 neighbors, uniform weight
Extra Trees Classifier	0.880	0.670	200 Estimators (best method in 2nd approach)
Extra Trees Classifier with PCA	0.733	0.639	200 Estimators with bootstrap on OOB error
Logistic Regression Classifier	0.774	0.588	L2 Penalty and 0.0001 tolerance
Logistic Regression Classifier with PCA	0.770	0.618	L2 Penalty and 0.0001 tolerance

I the first approach, XGBoost outperforms all the other algorithms, even without any parameter tuning. We therefore focus our attention on XGBoost.

In the second approach with over-sampling and PCA transform option, the Extra Trees Classifier performs better on non-PCA space.

6) XGBoost: Parameter tuning

XGBoost as an ideal algorithm for this problem – as a tree-based method it is insensitive to potential scaling or collinearity issues, and it is relatively robust to class imbalance. If tuned correctly, we think it can overcome over-fitting, even to individual subjects.

We therefore tried a few different ways to tune hyperparameters. We first tried tuning by hand, and then considered GridSearchCV and RandomizedSearchCV. However XGBoost has too many parameters and it is impossible to efficiently search this space using grid-based approaches.

We therefore tried using HyperOpt – a Bayesian optimization algorithm which updates its search of the parameter space based on performance. We used LeavePGroupsOut CV performance as the loss, but gains in performance were marginal. From this search we concluded that the following parameters worked fairly well:

- learning_rate = 0.01
- max_depth = 4
- min_child_weight = 5
- reg_lambda = 1
- reg_alpha = 0
- subsample = 0.8
- colsample_bylevel = 0.8
- colsample_bytree = 0.8
- gamma = 0

7) XGBoost: Feature reduction

Because all of our XGBoost classifiers were overfitting (as assessed by the wide gap in training versus validation performance) we sought to perform feature reduction. We took 3 steps:

a) *Reduce XYZ features using LDA* = We reasoned that if X, Y, and Z came from 222 sensors, it may be sufficient to reduce the 666 variables to 222 using dimensionality reduction. As opposed to PCA, which tries to preserve variance, we leveraged LDA which should preserve the dimension that maximizes discriminability between the two classes. We performed LDA on each set of 3 features.

b) *Compute ROC scores for each feature and eliminate low-scoring features* = We then sorted our features by their ROC value. We computed how well each (combined) feature by generating a ROC AUC score for each feature (we inverted any scores below 0.5). Features with a score less than 0.52 were eliminated, as we assumed that they would contribute little to the classifier.

c) *Prioritize predictive features (as assessed with ROC score) when removing collinear features* = Using these sorted ROC values, we then eliminated any features that had a correlation > 0.95 with a feature of higher ROC score.

Overall this left us with just 80 features. We then ran our parameter-tuned XGBoost model, which gave us our best overall performance on the Kaggle set.

8) Second Approach with over-sampling and PCA transform

In the second approach several methods were tested with and without PCA transform (200 components). The area under ROC curve is calculated and is used to measure the accuracy of different methods. The imbalanced dataset is balanced using MSMOTE method. Different class_0/class_1 ratios were tested and the ratio of 1 is considered.

However, the XGBoost method performs better in overall.

complete

November 7, 2018

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        import scipy.stats as st


        from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscrim
        from sklearn.pipeline import Pipeline
        from sklearn.model_selection import train_test_split, cross_val_score, RandomizedSearc
        from sklearn.model_selection import StratifiedKFold, GroupKFold, GridSearchCV, LeaveOn
        from sklearn.metrics import roc_auc_score, roc_curve
        from sklearn.ensemble import RandomForestClassifier

        from imblearn.over_sampling import SMOTENC

        import xgboost as xgb

        from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
        import time

        import string
```

0.1 1) Exploratory Data Analysis

Load and explore the data.

```
In [2]: train = pd.read_csv('train_data.csv')
        test = pd.read_csv('test_data.csv')

        train_labels = train['output']
```

```
In [4]: print(train.describe(include=np.object))
        print('\nOutput mean: %1.3f' % train['output'].mean())
        train.describe()
```

```
       subject state
count     4584  4584
```

```
unique      11     5
top          I     C
freq       663  2106

Output mean: 0.845
```

Out[4]:

	x1	x2	x3	x4	x5
count	4584.0	4584.000000	4584.000000	4584.000000	4584.000000
mean	0.0	0.000218	0.000218	0.625436	-270.199043
std	0.0	0.014770	0.014770	0.484063	163.971643
min	0.0	0.000000	0.000000	0.000000	-474.588020
25%	0.0	0.000000	0.000000	0.000000	-383.328439
50%	0.0	0.000000	0.000000	1.000000	-351.317222
75%	0.0	0.000000	0.000000	1.000000	-177.051209
max	0.0	1.000000	1.000000	1.000000	418.085156

	x6	x7	x8	x9	x10
count	4584.000000	4584.000000	4584.000000	4584.000000	4584.000000
mean	1.801323	-4.654634	253.735831	0.021039	-0.000004
std	1.143173	3.714967	147.405918	0.026384	0.000472
min	-0.237454	-19.295360	0.026692	0.005007	-0.004008
25%	0.815996	-6.708809	108.809493	0.007504	-0.000081
50%	1.578367	-3.324030	313.157669	0.011282	0.000003
75%	2.741501	-2.027700	370.832095	0.021861	0.000083
max	4.368496	6.250661	588.309282	0.423408	0.003588

	...	z215	z216	z217	z218
count	...	4584.000000	4584.000000	4584.000000	4584.000000
mean	...	-0.001628	0.001754	0.001374	0.001404
std	...	0.099405	0.070156	0.078443	0.091211
min	...	-3.805788	-0.999448	-1.886137	-1.264762
25%	...	-0.022183	-0.018118	-0.017379	-0.021389
50%	...	-0.000383	0.000673	0.000374	0.000507
75%	...	0.020556	0.020727	0.019355	0.022440
max	...	0.916725	0.708321	1.548114	2.949784

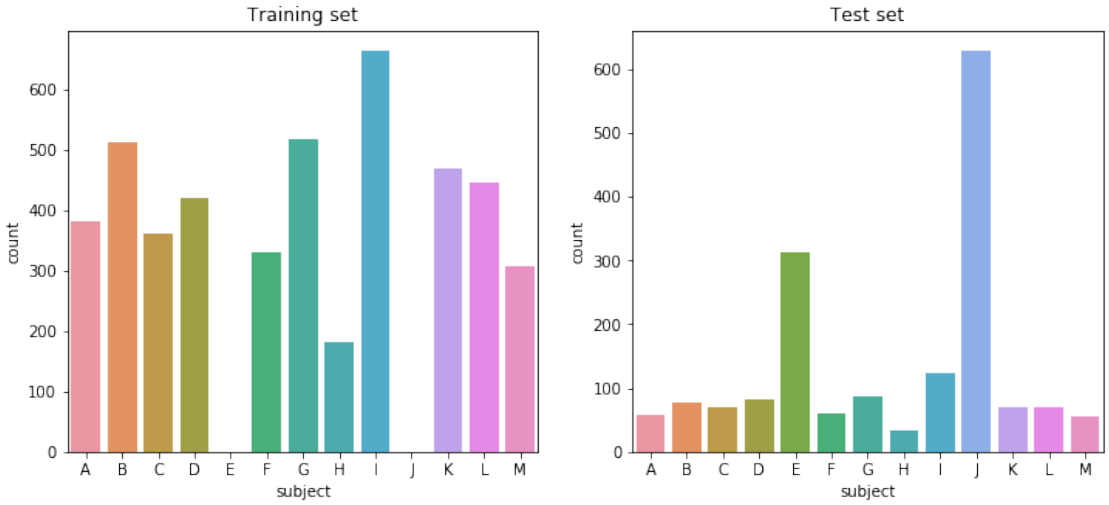
	z219	z220	z221	z222	phase
count	4584.000000	4584.000000	4584.000000	4584.000000	4584.000000
mean	-57.916190	96.554052	-52.630948	28.735112	2.519634
std	243.742549	597.912778	599.208382	217.842477	1.158998
min	-2182.646032	-6674.270678	-8311.046315	-3671.954955	1.000000
25%	-31.817506	-1.715924	-11.180312	-0.050475	1.000000
50%	-2.919292	0.098110	-0.492411	0.001721	3.000000
75%	-0.277689	13.684733	-0.032526	0.483496	4.000000
max	2408.234281	6111.797852	6412.126601	2816.376179	4.000000

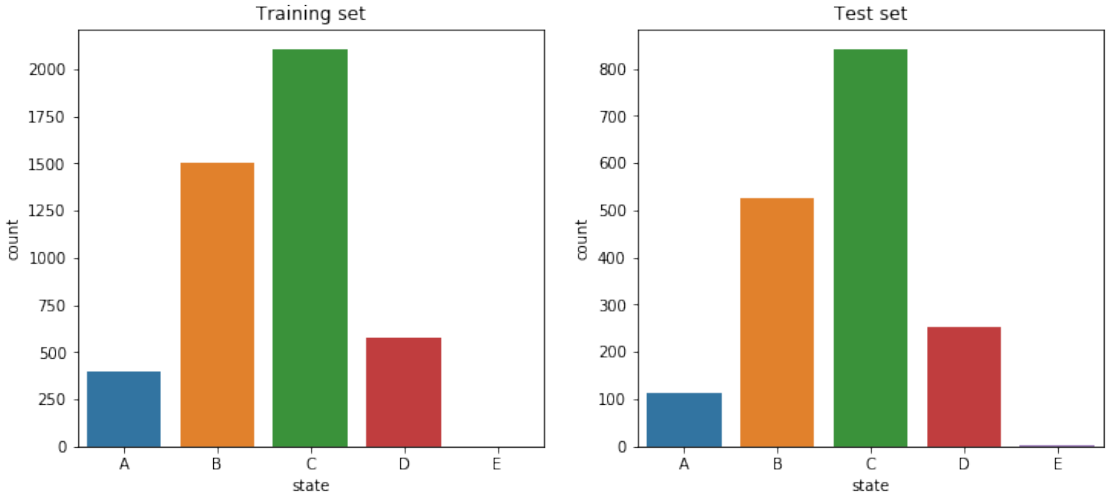
	output

```
count  4584.000000
mean      0.844895
std       0.362044
min       0.000000
25%       1.000000
50%       1.000000
75%       1.000000
max       1.000000

[8 rows x 668 columns]
```

We have a very unbalanced dataset, with 84.5% of points in class 1. There are 666 numeric features and 3 categorical features.

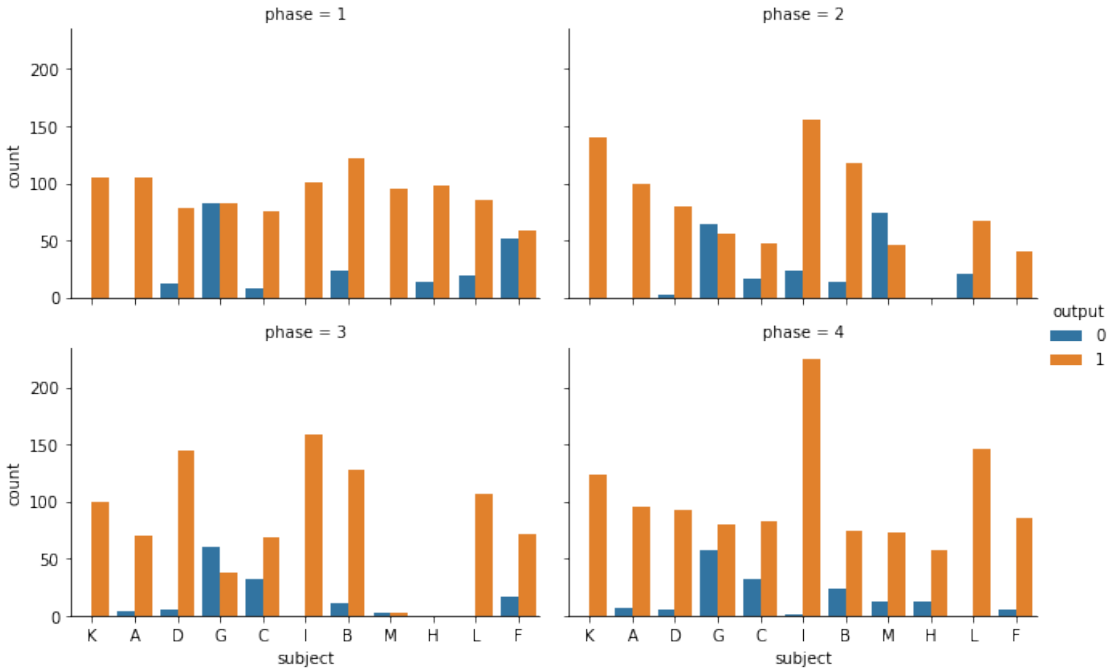
```
In [5]: # check overlap of categorical features
        tr_subjects = train['subject'].unique()
        tr_subjects.sort()
        te_subjects = test['subject'].unique()
        te_subjects.sort()
        plt.figure(figsize=(12, 5))
        plt.subplot(1,2,1)
        sns.countplot(x='subject', data=train, order=te_subjects)
        plt.title('Training set')
        plt.subplot(1,2,2)
        # plt.figure()
        sns.countplot(x='subject', data=test, order=te_subjects)
        plt.title('Test set')

        tr_states = train['state'].unique()
        tr_states.sort()
        te_states = test['state'].unique()
        te_states.sort()
        plt.figure(figsize=(12, 5))
        plt.subplot(1,2,1)
        sns.countplot(x='state', data=train, order=te_states)
        plt.title('Training set')
        plt.subplot(1,2,2)
        sns.countplot(x='state', data=test, order=te_states)
        plt.title('Test set')

Out[5]: Text(0.5,1,'Test set')
```





As demonstrated in Section, the phase variable is clearly categorical, so it may be better to use one-hot encoding instead.

```
In [6]: sns.catplot(x="subject", hue="output", col="phase", data=train, kind="count", col_wrap=
```



0.2 2) Feature cleaning, extraction, and engineering

- a) Separate binary and numeric features
- b) Remove low variance features
- c) Renormalize numeric features
- d) Remove features with strong collinearity
- e) Convert categorical features to one-hot-encoding

```
In [7]: # this functions "cleans" numeric features and one-hot encodes categoricals

        def clean_features(train_num, test_num, remove_corr=True, use_subject=True):

            # (2b) remove all features that show no variation in EITHER the training set OR in
            bad_features = (train_num.var()==0) | (test_num.var()==0)
            to_drop = train_num.columns[bad_features]
            train_num = train_num.drop(to_drop, axis=1)
            test_num = test_num.drop(to_drop, axis=1)

            # (2a) separate binary and putatively categorical features
            merged = pd.concat([train_num, test_num])
            binary = (merged.nunique()==2) & (merged.max()==1)  # don't normalize binary featu
            numeric = ~binary
```

```
    # (2c) rescale by training set mean and std
    means = train_num.loc[:,numeric].mean()
    stds = train_num.loc[:,numeric].std()
    train_num.loc[:,numeric] = (train_num.loc[:,numeric]-means) / stds
    test_num.loc[:,numeric] = (test_num.loc[:,numeric]-means) / stds

    # (2b) remove binary features that occur very rarely
    binary_freq = train_num.loc[:,binary].mean()
    binary_freq[binary_freq>0.5] = 1-binary_freq[binary_freq>0.5]
    to_drop = train_num.columns[binary][binary_freq<0.005]
    train_num = train_num.drop(to_drop, axis=1)
    test_num = test_num.drop(to_drop, axis=1)

    # (2d) remove features with strong collinearity (correlations)
    features_corr = abs(train_num.corr())
    features_corr = features_corr.where(np.triu(np.ones(features_corr.shape), k=1).asty
    to_drop = train_num.columns[features_corr.max()>0.95]
    train_num = train_num.drop(to_drop, axis=1)
    test_num = test_num.drop(to_drop, axis=1)

    # (2e) one-hot encoding of categorical variables
    if use_subject:
        categories = ['state','subject','phase']
    else:
        categories = ['state','phase']
    d = {1:'one', 2:'two', 3:'three', 4:'four'}
    merged = pd.concat([train[categories], test[categories]])
    merged['phase'] = merged['phase'].map(d)
    merged = pd.get_dummies(merged)
    if use_subject:
        merged.drop(['subject_E', 'subject_J'], axis=1, inplace=True)
    train_cat = merged.iloc[:train.shape[0], :]
    test_cat = merged.iloc[train.shape[0]+1:, :]

    if 'phase' in train_num.columns:
        train_num = train_num.drop('phase', axis=1) # redundant, though could leave ju
        test_num = test_num.drop('phase', axis=1)

    # merge features
    train_all = pd.concat([train_num, train_cat], axis=1)
    test_all = pd.concat([test_num, test_cat], axis=1)

    return train_all, test_all
```

In [8]:
```
# (2f) engineering deterministic interaction features

def make_interaction_features(train_all=None, test_all=None):
```

```
    # engineer new features
    binary = train_all.loc[:,train_all.nunique()==2]

    features = binary.columns
    nfeatures = len(features)

    train_new = None
    test_new = None

    # remove single deterministic features (e.g. subject_K)
    for f in features:
        temp = pd.concat([binary.loc[:,f], train_labels], axis=1)
        grouped = temp.groupby(f)
        good = ((grouped.mean()==1) | (grouped.mean()==0)) & (grouped.count()>80)
        if good['output'].any():
            print(f)
            binary = binary.drop(f, axis=1)

    # find pairs of binary features that reliably determine output
    for (i, f1) in enumerate(features):
        for j in range(i+1, nfeatures):
            f2 = features[j]
            temp = pd.concat([binary.loc[:,[f1,f2]], train_labels], axis=1)
            grouped = temp.groupby([f1,f2])
            good = ((grouped.mean()==1) | (grouped.mean()==0)) & (grouped.count()>80)
            if good['output'].any():
                print(f1,f2)
                if train_new is None:
                    train_new = pd.DataFrame(train_all.loc[:, [f1, f2]].any(axis=1), co
                    test_new  = pd.DataFrame(test_all.loc[:, [f1, f2]].any(axis=1), col
                else:
                    train_new[f1+'_&_'+f2] = train_all.loc[:, [f1, f2]].any(axis=1)
                    test_new[f1+'_&_'+f2]  = test_all.loc[:, [f1, f2]].any(axis=1)

    train_all = pd.concat([train_all, train_new], axis=1)
    test_all = pd.concat([test_all, test_new], axis=1)

    return train_all, test_all
```

In [9]:
```
# (2g) engineer combined X-Y-Z features using LDA

def make_LDA_features(train_all=None, test_all=None):

    # transform features using LDA on each set of three (x,y,z) features
    coords = ['x','y','z']
    lda = LinearDiscriminantAnalysis()

    train_new = None
```

```
            test_new = None

            for i in range(1,223):
                features = [c + str(i) for c in coords]
                try:
                    tr_new = lda.fit_transform(train[features], train['output'])
                    te_new = lda.transform(test[features])
                    mu = tr_new.mean()
                    sig = tr_new.std()
                    tr_new = (tr_new-mu)/sig
                    te_new = (te_new-mu)/sig

                    if train_new is None:
                        train_new = pd.DataFrame(tr_new, columns = ['n' + str(i)])
                        test_new  = pd.DataFrame(te_new, columns = ['n' + str(i)])
                    else:
                        train_new['n' + str(i)] = tr_new
                        test_new['n' + str(i)]  = te_new
                except: # skip any that fail
                    continue

            train_all = pd.concat([train_all, train_new], axis=1)
            test_all = pd.concat([test_all, test_new], axis=1)

            return train_all, test_all
```

```
In [10]: # (2h) prioritize predictive features using ROC

        def filter_by_roc(train_all, test_all):

            # sort features by roc
            rocs = train_all.apply(lambda x: roc_auc_score(train['output'], x))
            rocs[rocs<0.5] = 1-rocs[rocs<0.5]
            rocs.sort_values(ascending=False, inplace=True)

            # remove low-scoring features
            to_drop = rocs.index[rocs<0.52]
            rocs = rocs.drop(to_drop)

            # remove correlated features
            features_corr = abs(train_all[rocs.index].corr())
            features_corr = features_corr.where(np.triu(np.ones(features_corr.shape), k=1).as
            to_drop = rocs.index[features_corr.max()>0.95]
            rocs = rocs.drop(to_drop)

            train_all = train_all[rocs.index]
            test_all = test_all[rocs.index]
```

```
            return train_all, test_all, rocs
```

```
In [11]: # this code runs the functions
         train_num = train.select_dtypes(include='number')
         train_num = train_num.drop(columns='output')
         test_num = test.select_dtypes(include='number')

         # use original features
         train_all, test_all = clean_features(train_num, test_num)
```

0.3 3) Training, validation, cross-validation sets

Holdout a fraction of the training data (25%) as a validation set, to avoid overfitting as we play with different models. For the final model, we will use all of the data.

We know that the test set contains subjects that we have not seen during training ('E' and 'J'). To simulate this we can holdout a whole subjects data from the training set, and look at our performance on this heldout subject. It may be most helpful to remove a subject or two that show high similarity in their feature distributions to E and J.

```
In [12]: merged = pd.concat([train_all, test_all])
         merged['subject'] = pd.concat([train['subject'], test['subject']])

         group_means = merged.groupby('subject').mean()

         group_corr = group_means.transpose().corr()
         np.fill_diagonal(group_corr.values, np.nan)
         print('Most similar subjects:')
         holdouts = group_corr[['E','J']].idxmax()
         print(holdouts)
         group_corr
```

```
Most similar subjects:
subject
E    A
J    M
dtype: object
```

Out[12]:

subject	A	B	C	D	E	F	G	\
subject								
A	NaN	0.659936	0.487559	0.518157	0.651023	0.416629	0.438115	
B	0.659936	NaN	0.769963	0.628549	0.633389	0.655423	0.599244	
C	0.487559	0.769963	NaN	0.823189	0.382757	0.749086	0.672863	
D	0.518157	0.628549	0.823189	NaN	0.315304	0.687111	0.663163	
E	0.651023	0.633389	0.382757	0.315304	NaN	0.338465	0.418228	
F	0.416629	0.655423	0.749086	0.687111	0.338465	NaN	0.642792	
G	0.438115	0.599244	0.672863	0.663163	0.418228	0.642792	NaN	
H	0.370466	0.609356	0.626821	0.529636	0.346441	0.491959	0.523068	
I	0.435010	0.190157	0.291190	0.512003	0.207622	0.509172	0.416653	

```
J        0.309993  0.486348  0.755716  0.827313  0.077599  0.749586  0.688046
K        0.476423  0.730499  0.864556  0.792586  0.289190  0.787108  0.572255
L        0.627475  0.841944  0.790489  0.662255  0.574040  0.776638  0.706042
M        0.424063  0.599584  0.784299  0.777646  0.290846  0.793032  0.697917
```

subject	H	I	J	K	L	M
subject						
A	0.370466	0.435010	0.309993	0.476423	0.627475	0.424063
B	0.609356	0.190157	0.486348	0.730499	0.841944	0.599584
C	0.626821	0.291190	0.755716	0.864556	0.790489	0.784299
D	0.529636	0.512003	0.827313	0.792586	0.662255	0.777646
E	0.346441	0.207622	0.077599	0.289190	0.574040	0.290846
F	0.491959	0.509172	0.749586	0.787108	0.776638	0.793032
G	0.523068	0.416653	0.688046	0.572255	0.706042	0.697917
H	NaN	0.203617	0.467167	0.574195	0.581269	0.538094
I	0.203617	NaN	0.481760	0.328824	0.367589	0.432588
J	0.467167	0.481760	NaN	0.802728	0.585289	0.844770
K	0.574195	0.328824	0.802728	NaN	0.747856	0.808832
L	0.581269	0.367589	0.585289	0.747856	NaN	0.727748
M	0.538094	0.432588	0.844770	0.808832	0.727748	NaN

From this basic analysis it appears that E is most similar to A, whereas J shows high similarity to multiple subjects, including M, K, and D.

It may therefore be most instructive to hold out data from A and M.

In [13]:
```
# holdout_subjects = train['subject'].isin(holdouts)
holdout_subjects = train['subject'].isin(['A','M'])

frac = 0.25

# validation set with all subjects
X_train1, X_test1, y_train1, y_test1 = train_test_split(train_all, train_labels, test_
groups_train1 = train['subject'].loc[X_train1.index]
groups_test1 = train['subject'].loc[X_test1.index]

# validation set without subjects A or M
frac2 = (frac - np.mean(holdout_subjects)) * len(holdout_subjects) / np.sum(~holdout_s
X_train2, X_test2, y_train2, y_test2 = train_test_split(train_all.loc[~holdout_subject
                                                        train_labels.loc[~holdout_subj
                                                        test_size=frac2, random_state=
X_test2 = pd.concat([X_test2, train_all.loc[holdout_subjects,:]])
y_test2 = pd.concat([y_test2, train_labels.loc[holdout_subjects]])

groups_train2 = train['subject'].loc[X_train2.index]
groups_test2 = train['subject'].loc[X_test2.index]
```

0.3.1 Cross validation

For the second set, it may be instructive to try a few different ways of splitting: one with a random set of 5 folds, another with 5 different subjects held out. We will use sklearn's StratifiedKFold (to preserve class imbalance) for subject-independent splitting and GroupKFold for subject-dependent splitting. Lastly, we use LeavePGroupsOut(2) to measure performance across all situations with 2 new test subjects.

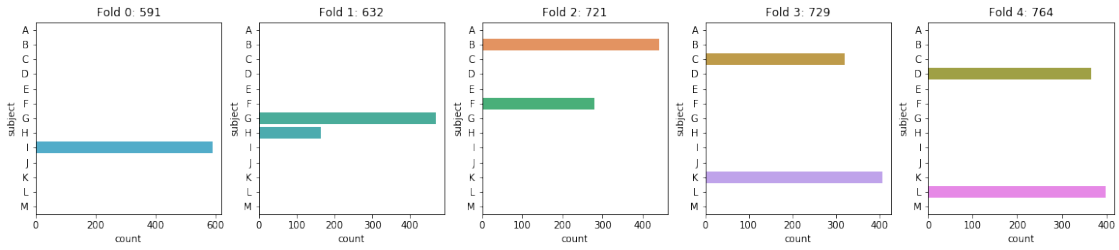
```
In [14]: SEED = 100 # for reproducibility
         NFOLDS = 5 # set folds for out-of-fold prediction

         skf = StratifiedKFold(NFOLDS, shuffle=True, random_state=SEED)
         gkf = GroupKFold(NFOLDS)
         lpgo = LeavePGroupsOut(n_groups=2)
```

```
In [15]: # fraction of class 1 in each fold
         [y_train2[test_index].mean() for (train_index, test_index) in gkf.split(X_train2, y_t
```

```
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/pandas/core/series.py:842: FutureWar
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-listlike
  return self.loc[key]
```

```
Out[15]: [0.8190045248868778,
          0.8477366255144033,
          0.8460076045627376,
          0.8309352517985612,
          0.8620689655172413]
```

```
In [16]: # plot to show groups in each fold
         plt.figure(figsize=(20, 8))
         for (i, (train_index, test_index)) in enumerate(gkf.split(X_train2, y_train2, groups_t
             fold = groups_train2.iloc[test_index]
             plt.subplot(2,5,i+1)
             sns.countplot(y=fold, order=te_subjects)
             plt.title('Fold %d: %d' % (i, fold.shape[0]))
```



0.4 4) optional code for resampling to overcome class imbalance

SMOTE oversamples the minority class to mitigate class imbalance. I do not use it for XGBoost as it does not seem to improve performance.

```
In [ ]: # smote = SMOTENC(categorical_features=np.where(X_train1.nunique()<=4)[0])

        # X_temp, y_train1 = smote.fit_sample(X_train1, y_train1)
        # X_train1 = pd.DataFrame(X_temp, columns=X_train1.columns)
        # X_train1 = X_train1.apply(lambda x: pd.to_numeric(x, errors='ignore'))
        # groups_train1 = X_train1['subject_'+tr_subjects].idxmax(axis=1).map(lambda x: x[-1])

        # X_temp, y_train2 = smote.fit_sample(X_train2, y_train2)
        # X_train2 = pd.DataFrame(X_temp, columns=X_train2.columns)
        # X_train2 = X_train2.apply(lambda x: pd.to_numeric(x, errors='ignore'))
        # X_train2[np.isnan(X_train2)] = 0
        # groups_train2 = X_train2['subject_'+tr_subjects].idxmax(axis=1).map(lambda x: x[-1])
```

0.5 5) Exploring different classifiers

Helper functions for testing different classifiers

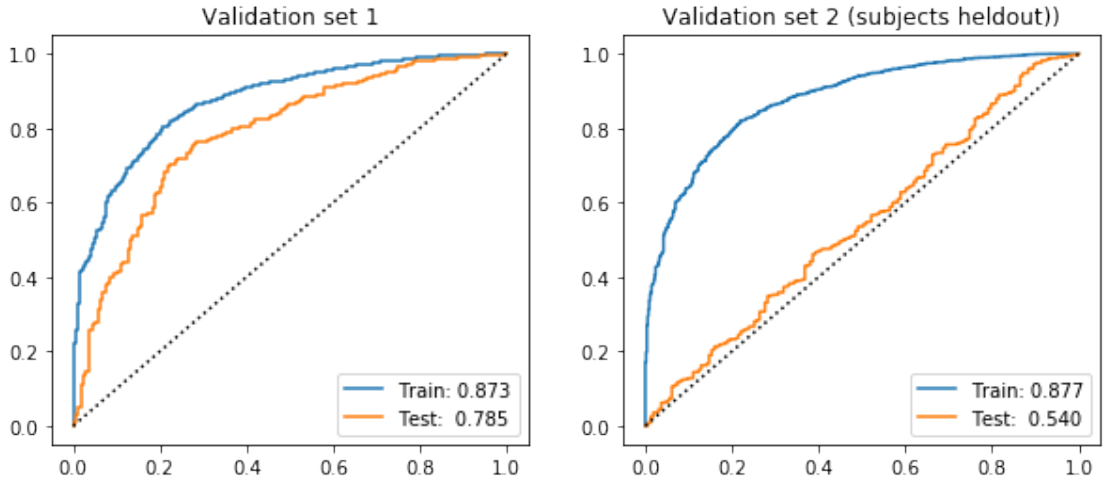
```
In [17]: def plotROC(y_test, scores_test, y_train=None, scores_train=None):
             auc = []
             if y_train is not None:
                 fpr, tpr, thresholds = roc_curve(y_train, scores_train)
                 auc.append(roc_auc_score(y_train, scores_train))
                 plt.plot(fpr,tpr,label='Train: %1.3f' % auc[0])

             fpr, tpr, thresholds = roc_curve(y_test, scores_test)
             auc.append(roc_auc_score(y_test, scores_test))
             plt.plot(fpr,tpr,label='Test:  %1.3f' % auc[-1])
             plt.plot([0,1], [0,1], ':k')
             plt.legend(loc='lower right')
             return auc


         # given a pipeline, fit and generate roc values and plots for both validation sets
         # if pipe2 is given, assume both pipes are already fit
         def predictAll(pipe1, pipe2=None):

             fit = pipe2 is None

             if fit:
                 pipe1.fit(X_train1, y_train1)
             scores_train = pipe1.predict_proba(X_train1)
             scores_test = pipe1.predict_proba(X_test1)

             plt.figure(figsize=(10, 4))
```

```
    plt.subplot(1,2,1)
    auc1 = plotROC(y_test1, scores_test[:,1], y_train1, scores_train[:,1])
    plt.title('Validation set 1')

    if fit:
        pipe1.fit(X_train2, y_train2)
        scores_train = pipe1.predict_proba(X_train2)
        scores_test = pipe1.predict_proba(X_test2)
    else:
        scores_train = pipe2.predict_proba(X_train2)
        scores_test = pipe2.predict_proba(X_test2)

    plt.subplot(1,2,2)
    auc2 = plotROC(y_test2, scores_test[:,1], y_train2, scores_train[:,1])
    plt.title('Validation set 2 (subjects heldout))')

    # cross-validation performance
    auc3 = cross_val_score(pipe1, train_all, y=train_labels, groups=train['subject'],
    auc4 = cross_val_score(pipe1, train_all, y=train_labels, groups=train['subject'],
    auc5 = cross_val_score(pipe1, train_all, y=train_labels, groups=train['subject'],
    print('SKF:  %1.3f' % auc3)
    print('GKF:  %1.3f' % auc4)
    print('LPGO: %1.3f' % auc5)

    return auc1, auc2, auc3, auc4, auc5
```

0.5.1 Linear Discriminant Analysis

```
In [18]: estimators = []
         estimators.append(('LDA', LinearDiscriminantAnalysis(solver='svd', shrinkage=None, n_
         ldapipe = Pipeline(estimators)
         predictAll(ldapipe);
```

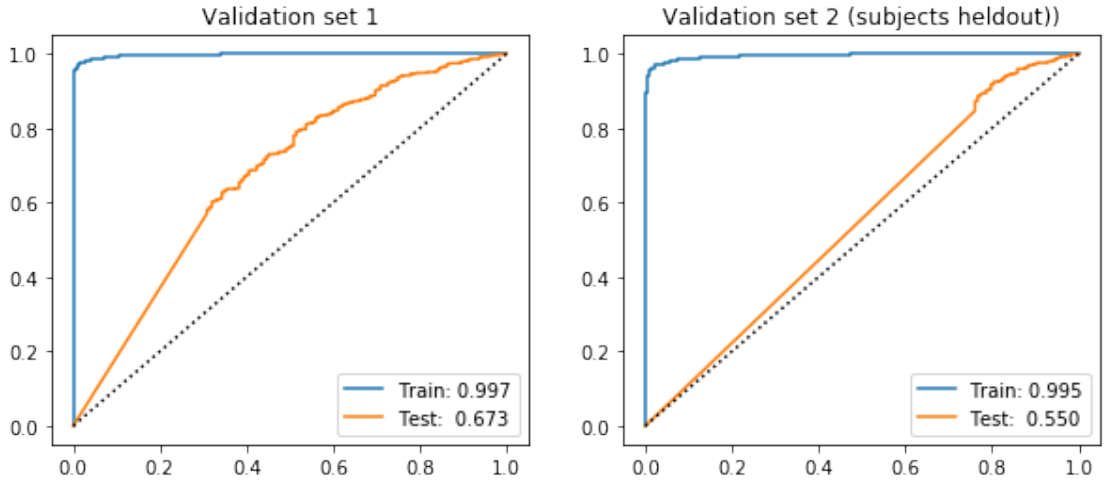
```
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")


SKF:  0.795
GKF:  0.494
LPGO: 0.505
```



0.5.2 Quadratic Discriminant Analysis

```
In [19]: estimators = []
         estimators.append(('QDA', QuadraticDiscriminantAnalysis() ))

         qdapipe = Pipeline(estimators)
         predictAll(qdapipe);
```

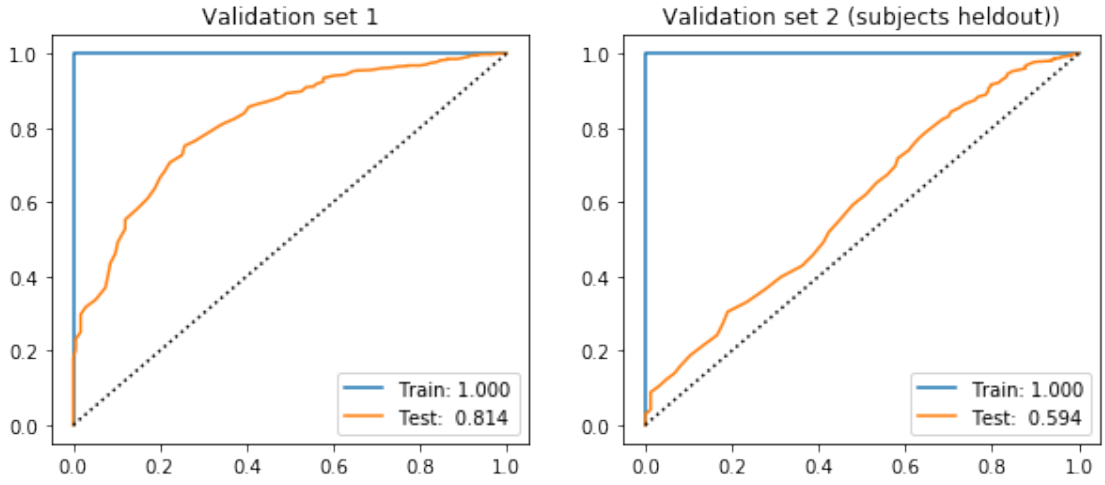
```
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:686
  warnings.warn("Variables are collinear")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:686
  warnings.warn("Variables are collinear")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:686
  warnings.warn("Variables are collinear")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:686
  warnings.warn("Variables are collinear")


SKF:  0.654
GKF:  0.493
LPGO: 0.567
```



0.5.3 Random Forests

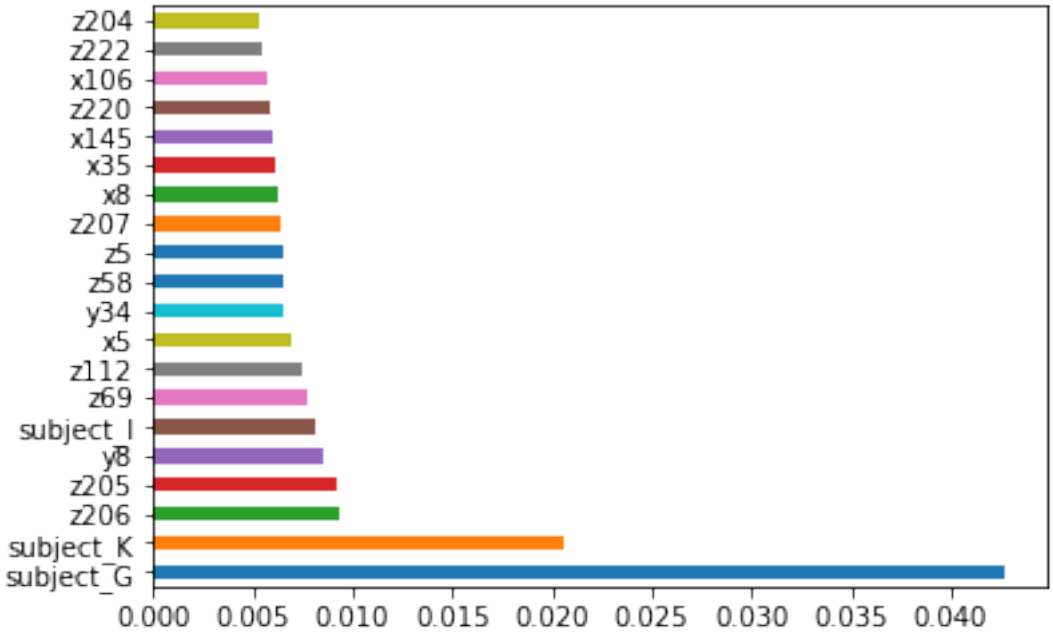
```
In [20]: estimators = []
         estimators.append(('RF', RandomForestClassifier(n_estimators=200,
                                                         oob_score=True,
                                                         class_weight='balanced',
                                                         random_state=0,
                                                         n_jobs=-1) ))

         rfpipe = Pipeline(estimators)
         predictAll(rfpipe);
```

```
SKF:  0.820
GKF:  0.560
LPGO: 0.536
```



```
In [28]: rfpipe.fit(X_train2, y_train2)
         best_features = pd.Series(rfpipe.named_steps['RF'].feature_importances_, index=train_a
         best_features.iloc[:20].plot(kind='barh')
```

```
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7f201f51e320>
```



0.5.4 XGBoost

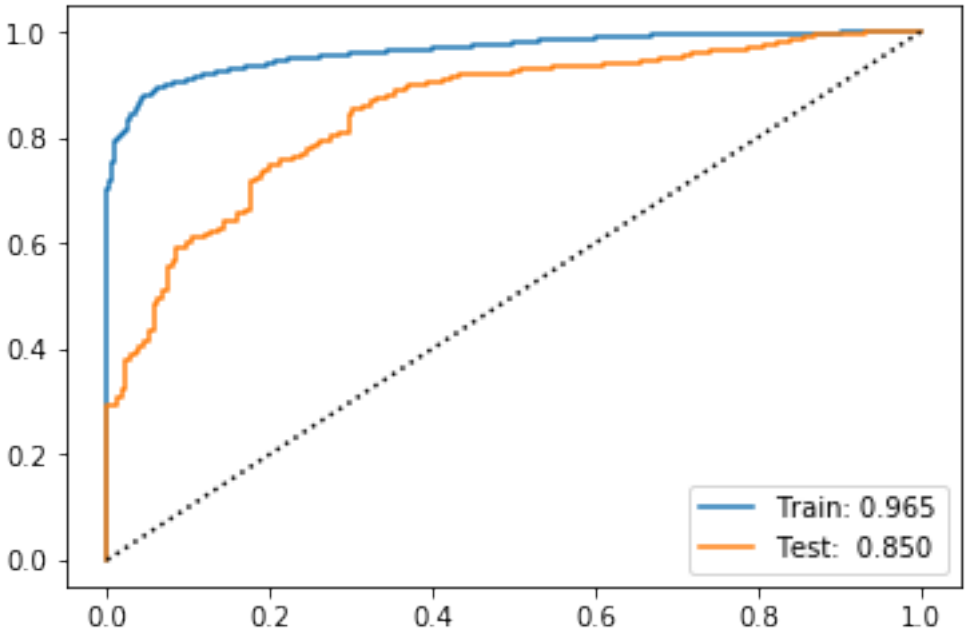
```
In [22]: # Rashmi's parameters
         xg = xgb.XGBClassifier(seed=0, n_estimators=1000, max_depth=3, colsample_bylevel=0.8,
                                colsample_bytree=0.7,learning_rate=0.01, reg_lambda=0.1 ,
                                scale_pos_weight = 0.18357862, nthread=-1, n_jobs=-1)
         xg
```

```
Out[22]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=0.8,
                colsample_bytree=0.7, gamma=0, learning_rate=0.01, max_delta_step=0,
                max_depth=3, min_child_weight=1, missing=None, n_estimators=1000,
                n_jobs=-1, nthread=-1, objective='binary:logistic', random_state=0,
                reg_alpha=0, reg_lambda=0.1, scale_pos_weight=0.18357862, seed=0,
                silent=True, subsample=1)
```

```
In [29]: xg.fit(X_train1, y_train1, eval_metric='auc', eval_set=[(X_test1, y_test1)], early_sto
         scores_train = xg.predict_proba(X_train1)
```

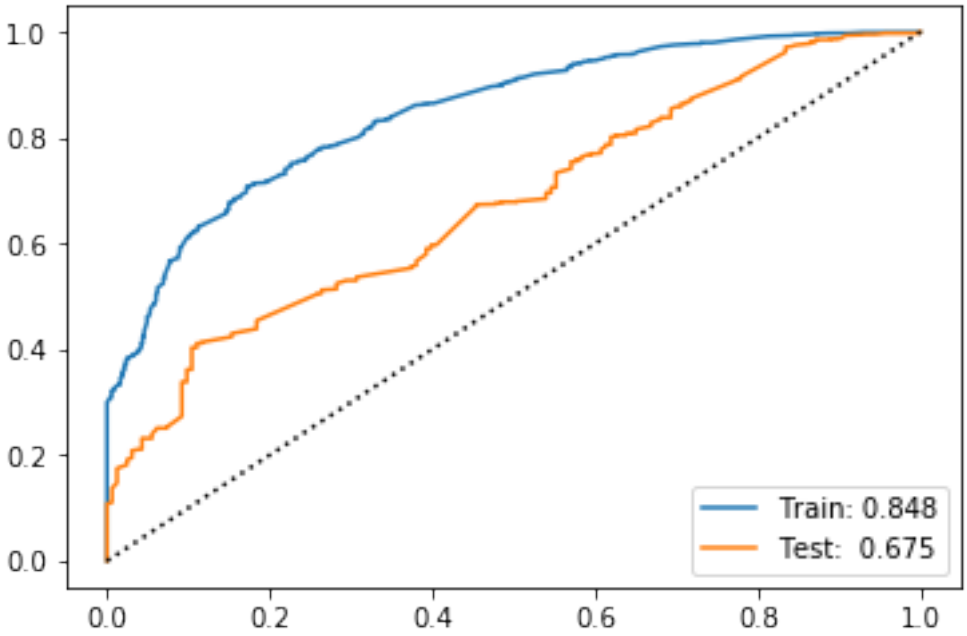
```
scores_test = xg.predict_proba(X_test1)

plotROC(y_test1, scores_test[:,1], y_train1, scores_train[:,1])
```

Out[29]: [0.964508522172048, 0.849804325437693]



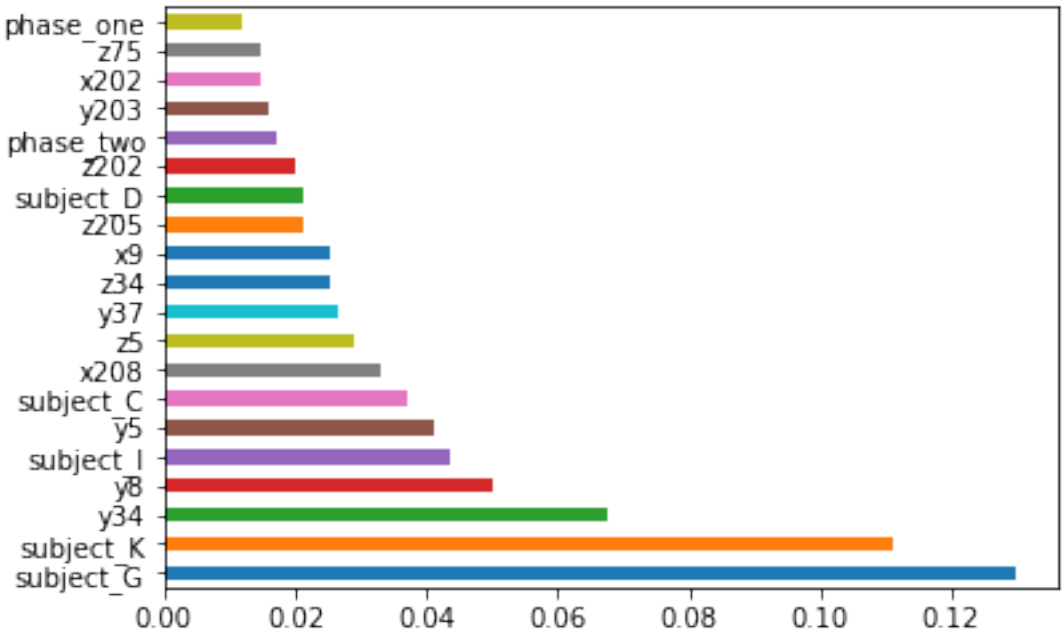
In [30]:
```
xg.fit(X_train2, y_train2, eval_metric='auc', eval_set=[(X_test2, y_test2)], early_st
scores_train = xg.predict_proba(X_train2)
scores_test = xg.predict_proba(X_test2)

plotROC(y_test2, scores_test[:,1], y_train2, scores_train[:,1])
```

Out[30]: [0.8484157122536276, 0.6746377624819192]




```
In [32]: best_features = pd.Series(xg.feature_importances_,index=train_all.columns).sort_values
         best_features.iloc[:20].plot(kind='barh')
```

```
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x7f201f573c18>
```



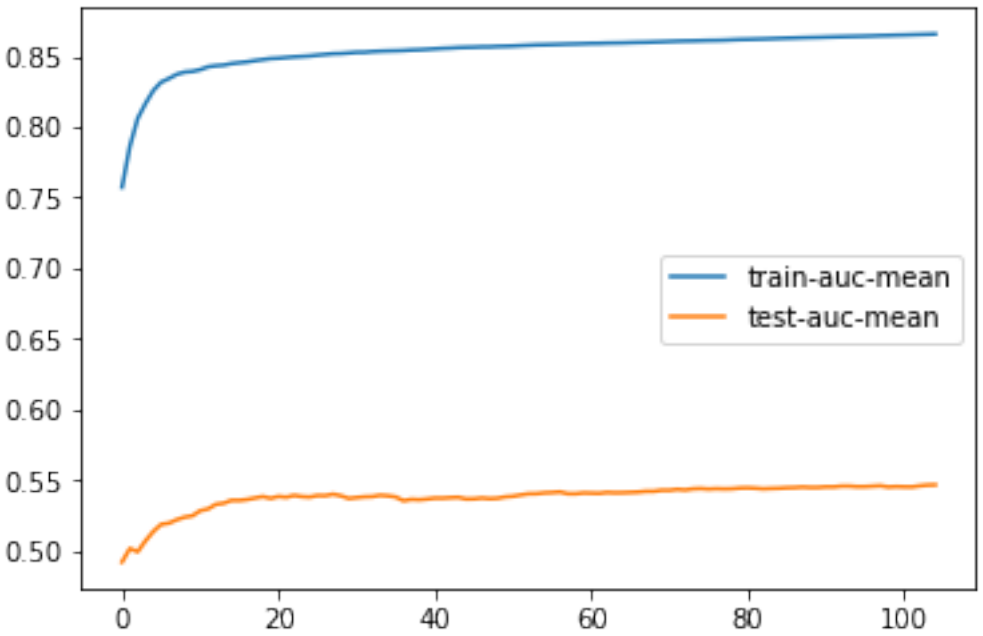
Many of the "best" features identified by XGBoost overlap with those used by Random Forests – this should not surprise us as they are both tree-based algorithms.

Use cross-validation to determine optimal number of trees.

In [31]:
```
# evaluate performance using LPGO cross-validation
folds = list(lpgo.split(train_all, train_labels, train['subject']))

param = xg.get_xgb_params()
xgtrain = xgb.DMatrix(train_all, label=train_labels)

cvresult = xgb.cv(param, xgtrain, num_boost_round=1000, folds=folds, metrics='auc', ea
print(cvresult['test-auc-mean'].iloc[-1])
cvresult.plot(y=['train-auc-mean','test-auc-mean'])
```

```
0.5467649818181818
```

Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7f20207cf438>



In [33]:
```
# first submission: just use the parameters Rashmi gave in Section, directly on our c
xg.set_params(n_estimators=cvresult.shape[0])
xg.fit(train_all, train_labels, eval_metric='auc', verbose=False)
scores_train = xg.predict_proba(train_all)
print(roc_auc_score(train_labels, scores_train[:,1]))
scores = xg.predict_proba(test_all)
```

```
submithis = pd.DataFrame([test_all.index,scores[:,1]]).T
submithis.columns = ['id','output']
submithis.id = submithis.id.astype(int)
# submithis.to_csv('submitthis_xgb1nosubjects.csv',index=False)  # submission 1: 0.71
```

```
0.8367031593457972
```

0.6 6) Tuning XGBoost parameters

0.6.1 Manual tuning

We first tried tuning parameters manually, knowing that we need to increase regularization to reduce overfitting.

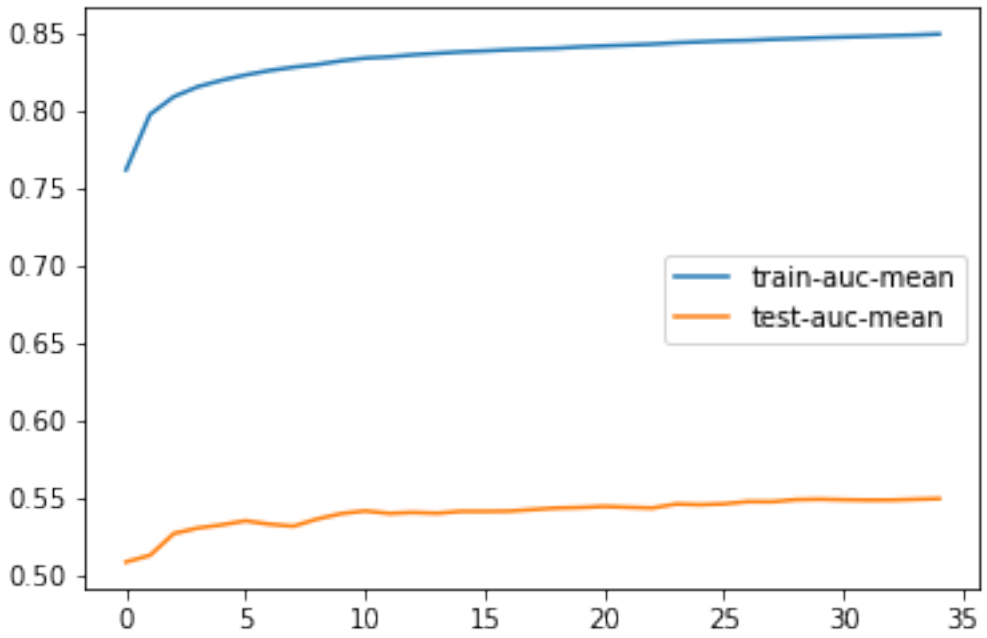
In [34]:
```
# now trying to tune parameters using cross-validation:
xg = xgb.XGBClassifier(random_state=100, n_jobs=-1, nthread=11, scale_pos_weight = 0.
                       # number of trees and learning rate
                       n_estimators=1000,
                       learning_rate=0.01,
                       base_score=train_labels.mean(),
                       # tree parameters
                       max_depth=4,
                       min_child_weight=5,
                       gamma=0,
                       # regularization
                       reg_lambda=1,
                       # these parameters add randomness (decrease to reduce overfitt
                       subsample=0.8,
                       colsample_bylevel=0.8, # 0.8,
                       colsample_bytree=0.8)
xg
```

Out[34]:
```
XGBClassifier(base_score=0.8448952879581152, booster='gbtree',
       colsample_bylevel=0.8, colsample_bytree=0.8, gamma=0,
       learning_rate=0.01, max_delta_step=0, max_depth=4,
       min_child_weight=5, missing=None, n_estimators=1000, n_jobs=-1,
       nthread=11, objective='binary:logistic', random_state=100,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=0.18357862, seed=None,
       silent=True, subsample=0.8)
```

In [35]:
```
# evaluate performance using LPGO cross-validation
folds = list(lpgo.split(train_all, train_labels, train['subject']))

param = xg.get_xgb_params()
xgtrain = xgb.DMatrix(train_all, label=train_labels)

cvresult = xgb.cv(param, xgtrain, num_boost_round=1000, folds=folds, metrics='auc', ea
print(cvresult['test-auc-mean'].iloc[-1])
cvresult.plot(y=['train-auc-mean','test-auc-mean'])
```

```
0.5489839272727273
```

Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x7f201f771860>



0.6.2 RandomizedSearchCV

```
In [ ]: # modified from http://danielhnyk.cz/how-to-use-xgboost-in-python/
        one_to_left = st.beta(10, 1)
        from_zero_positive = st.expon(0, 50)

        params = {
            "learning_rate": st.uniform(0.01, 0.2),
            "n_estimators": st.randint(10, 200),
            "max_depth": st.randint(3, 20),
            "gamma": st.uniform(0, 10),
            'reg_lambda': st.uniform(0, 10),
            "min_child_weight": st.uniform(0, 10),
            "subsample": st.uniform(0.5, 0.9),
            "colsample_bytree": st.uniform(0.5, 0.9),
        }

        xg = xgb.XGBClassifier(random_state=0, n_jobs=-1, nthread=11, scale_pos_weight = 0.183
                               base_score=train_labels.mean())
        xg
```

```
In [ ]: %time
        # very slow!
        gs = RandomizedSearchCV(xg, params, n_jobs=-1, scoring='roc_auc', cv=lpgo, n_iter=200,
        gs.fit(X_train2, y_train2, groups_train2, eval_metric='auc', eval_set())

In [ ]: xg = gs.best_estimator_
        import pickle
        filename = 'xgb_randcv.sav'
        pickle.dump(gs, open(filename, 'wb'))

In [ ]: # evaluate performance using LPGO cross-validation
        folds = list(lpgo.split(train_all, train_labels, train['subject']))

        param = xg.get_xgb_params()
        xgtrain = xgb.DMatrix(train_all, label=train_labels)

        cvresult = xgb.cv(param, xgtrain, num_boost_round=1000, folds=folds, metrics='auc', ea
        print(cvresult['test-auc-mean'].iloc[-1])
        cvresult.plot(y=['train-auc-mean','test-auc-mean'])
```

0.6.3 Hyperopt

Bayesian optimization for "smart" searching of parameter space.

```
In [ ]: folds = list(lpgo.split(train_all, train_labels, train['subject']))

        xgtrain = xgb.DMatrix(train_all, label=train_labels)

In [ ]: def objective(params):
            print("Training with params: ")
            print(params)
            start = time.time()
            cv_results = xgb.cv(params, xgtrain, num_boost_round=2000,
                                folds=folds, metrics='auc', early_stopping_rounds=100)

            best_score = cv_results['test-auc-mean'].max()
            params['n_estimators'] = cv_results.shape[0]

            print("\tScore %1.3f (%d rounds in %2.1f s)\n\n" % (best_score, cv_results.shape[0]

            loss = 1-best_score

            return {'loss': loss, 'params': params, 'status': STATUS_OK}

In [ ]: space = {
            # constant parameters
            'booster': 'gbtree' ,
            'objective': 'binary:logistic',
            'eval_metric': 'auc',
```

```
        'base_score': train_labels.mean(),
        'scale_pos_weight': 1/train_labels.mean()-1,
        'silent': 1,
        'seed': 0,
        'nthread': 11,
        # variable parameters
        'learning_rate': hp.loguniform('learning_rate', np.log(0.005), np.log(0.2)),
        'max_depth': hp.choice('max_depth', np.arange(3, 10, dtype=int)),
        'min_child_weight': hp.uniform('min_child_weight', 1, 6),
        'subsample': hp.uniform('subsample', 0.5, 1),
        'gamma': hp.uniform('gamma', 0, 1),
        'colsample_bytree': hp.uniform('colsample_bytree', 0.5, 1),
        'reg_alpha': 0,
        'reg_lambda': hp.loguniform('reg_lambda', np.log(0.01), np.log(20)),
    }
```

In [ ]:
```
%time
# very slow!!
trials = Trials()
best = fmin(objective, space, algo=tpe.suggest, max_evals=200, trials=trials)
```

In [ ]:
```
# best parameters
xg = xgb.XGBClassifier(random_state=100, n_jobs=-1, nthread=11, scale_pos_weight = 0.18
                       # number of trees and learning rate
                       n_estimators=1000,
                       learning_rate=0.01,
                       base_score=train_labels.mean(),
                       # tree parameters
                       max_depth=4,
                       min_child_weight=5,
                       gamma=0,
                       # regularization
                       reg_lambda=1,
                       # these parameters add randomness (decrease to reduce overfitti
                       subsample=0.8,
                       colsample_bylevel=0.8, # 0.8,
                       colsample_bytree=0.8)
xg
```

In [ ]:
```
# evaluate performance using LPGO cross-validation
folds = list(lpgo.split(train_all, train_labels, train['subject']))

param = xg.get_xgb_params()
xgtrain = xgb.DMatrix(train_all, label=train_labels)

cvresult = xgb.cv(param, xgtrain, num_boost_round=1000, folds=folds, metrics='auc', ea
print(cvresult['test-auc-mean'].iloc[-1])
cvresult.plot(y=['train-auc-mean','test-auc-mean'])
```

In [ ]:
```
xg.set_params(n_estimators=cvresult.shape[0])
xg.fit(train_all, train_labels, eval_metric='auc')
scores_train = xg.predict_proba(train_all)
print(roc_auc_score(train_labels, scores_train[:,1]))
scores = xg.predict_proba(test_all)
submithis = pd.DataFrame([test_all.index,scores[:,1]]).T
submithis.columns = ['id','output']
submithis.id = submithis.id.astype(int)

# submithis.to_csv('submitthis_xgb3.csv',index=False)  # submission 3 on kaggle
```

0.7 7) XGBoost: Feature reduction

The biggest gains we achieved with XGBoost was not with parameter tuning but feature reduction.

In [36]:
```
# make new LDA features
train_new, test_new = make_LDA_features()

# clean them
train_all, test_all = clean_features(train_new, test_new, remove_corr=False)

# filter by roc
train_all, test_all, rocs = filter_by_roc(train_all, test_all)
```

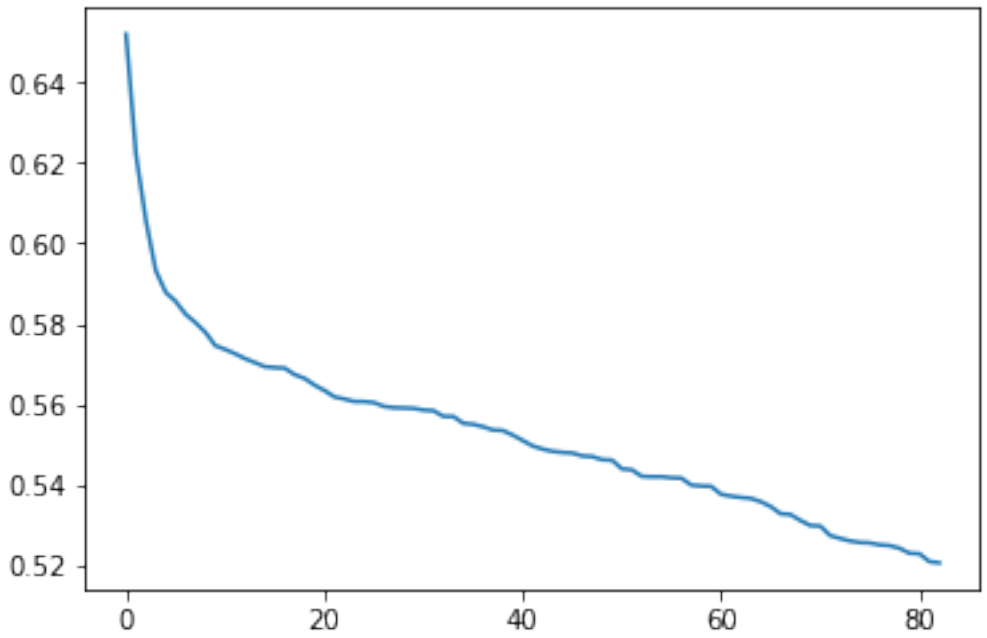
```
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
```

```
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
```

```
/home/gpho/anaconda3/envs/e82/lib/python3.7/site-packages/sklearn/discriminant_analysis.py:388
  warnings.warn("Variables are collinear.")
```

```
In [37]: plt.plot(rocs.values)
         rocs.head(30)
```

```
Out[37]: subject_G    0.652031
         n205         0.622157
         n206         0.605744
         n69          0.593219
         n204         0.587843
         n35          0.585592
         n8           0.582390
         n70          0.580311
         n30          0.577898
         n88          0.574586
         n21          0.573635
         n58          0.572557
         n34          0.571350
         n6           0.570359
         n207         0.569344
         n59          0.569112
         n89          0.568988
         n5           0.567365
         n108         0.566435
         subject_I    0.564784
         n157         0.563446
         n145         0.561845
         n60          0.561314
         n84          0.560694
         n20          0.560666
         subject_K    0.560418
         n147         0.559459
         n106         0.559162
         n146         0.559076
         n67          0.558990
         dtype: float64
```

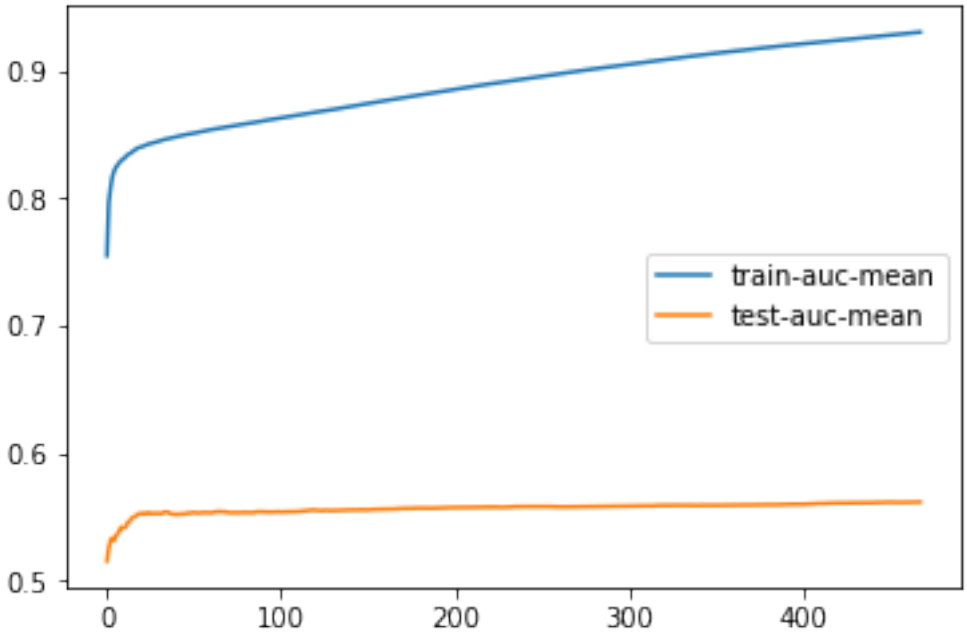


Using a reduced set of 83 features, let's retry our tuned XGBoost model:

```
In [38]: # evaluate performance using LPGO cross-validation
         folds = list(lpgo.split(train_all, train_labels, train['subject']))

         param = xg.get_xgb_params()
         xgtrain = xgb.DMatrix(train_all, label=train_labels)

         cvresult = xgb.cv(param, xgtrain, num_boost_round=1000, folds=folds, metrics='auc', ea
         print(cvresult['test-auc-mean'].iloc[-1])
         cvresult.plot(y=['train-auc-mean','test-auc-mean'])
```

```
0.5614987818181817
```

```
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x7f201f7e3780>
```




```
In [39]: xg.set_params(n_estimators=cvresult.shape[0])
         xg.fit(train_all, train_labels, eval_metric='auc')
         scores_train = xg.predict_proba(train_all)
         print(roc_auc_score(train_labels, scores_train[:,1]))
         scores = xg.predict_proba(test_all)
         submithis = pd.DataFrame([test_all.index,scores[:,1]]).T
         submithis.columns = ['id','output']
         submithis.id = submithis.id.astype(int)

         # submithis.to_csv('submitthis_xgb3.csv',index=False)  # submission 3 on kaggle
```

```
0.9174900125394787
```

1 Second Approach: Over-Sampling and PCA Transform

In this approach, over-sampling of training data is considered given then unbalanced distribution of output classes. The Modified synthetic minority oversampling technique (MSMOTE) is used. Some steps for the data cleaning from the first approach is applied in this approach again,

First, the required libraries are imported:

```
In [11]: import time
         import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
```

```
        from sklearn.svm import SVC
        from sklearn.svm import NuSVC
        from sklearn.linear_model import LogisticRegression
        from sklearn.gaussian_process import GaussianProcessClassifier
        from sklearn.ensemble import AdaBoostClassifier
        from sklearn.naive_bayes import GaussianNB
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.ensemble import ExtraTreesClassifier
        from sklearn.pipeline import Pipeline
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import roc_auc_score, roc_curve

        from imblearn.over_sampling import SMOTE
        from sklearn.decomposition import PCA

        def warn(*args, **kwargs):
            pass
        import warnings
        warnings.warn = warn

        print("Importing Lib Done!")
```

```
Importing Lib Done!
```

The training and test data are loaded again,

```
In [12]: train = pd.read_csv('train_data.csv')
         test = pd.read_csv('test_data.csv')
```

As mentioned above, the phase variable is categorical. Similar data cleaning is used below,

```
In [13]: # all numeric features
         train_num = train.select_dtypes(include='number')
         train_num = train_num.drop(columns='output')
         test_num = test.select_dtypes(include='number')

         # remove all features that show no variation in EITHER the training set OR in the tes
         bad_features = (train_num.var()==0) | (test_num.var()==0)
         to_drop = train_num.columns[bad_features]
         train_num = train_num.drop(to_drop, axis=1)
         test_num = test_num.drop(to_drop, axis=1)

         # separate binary and putatively categorical features
         merged = pd.concat([train_num, test_num])
         binary = (merged.nunique()==2) & (merged.max()==1)  # don't normalize binary features
         numeric = ~binary
```

```
        # rescale by training set mean and std
        means = train_num.loc[:,numeric].mean()
        stds = train_num.loc[:,numeric].std()
        train_num.loc[:,numeric] = (train_num.loc[:,numeric]-means) / stds
        test_num.loc[:,numeric] = (test_num.loc[:,numeric]-means) / stds

        # remove binary features that occur very rarely
        binary_freq = train_num.loc[:,binary].mean()
        binary_freq[binary_freq>0.5] = 1-binary_freq[binary_freq>0.5]
        to_drop = train_num.columns[binary][binary_freq<0.005]
        train_num = train_num.drop(to_drop, axis=1)
        test_num = test_num.drop(to_drop, axis=1)

        # remove features with strong collinearity (correlations)
        features_corr = abs(train_num.corr())
        features_corr = features_corr.where(np.triu(np.ones(features_corr.shape), k=1).astype
        to_drop = train_num.columns[features_corr.max()>0.95]
        train_num = train_num.drop(to_drop, axis=1)
        test_num = test_num.drop(to_drop, axis=1)

        # one-hot encoding of categorical variables
        categories = ['state','subject','phase']
        #categories = ['state','phase']
        d = {1:'one', 2:'two', 3:'three', 4:'four'}
        merged = pd.concat([train[categories], test[categories]])
        merged['phase'] = merged['phase'].map(d)
        merged = pd.get_dummies(merged)

        train_cat = merged.iloc[:train.shape[0], :]
        test_cat = merged.iloc[train.shape[0]:, :]
        train_num = train_num.drop('phase', axis=1)
        test_num = test_num.drop('phase', axis=1)

        # merge features
        train_all = pd.concat([train_num, train_cat], axis=1)
        train_labels = train['output']
        test_all = pd.concat([test_num, test_cat], axis=1)
```

The following function is defined for over-sampling of training data to account for imbalanced data,

```
In [14]: def overSample(data, label, randState, ratio):
             """Function to over-sample the imbalanced training data
                ratio: ratio of class 0 to class 1
             """
             sm = SMOTE(random_state=randState, ratio = ratio)
             xx, xx_label = sm.fit_sample(data, label)
             xx = pd.DataFrame(xx, index=[i for i in range(len(xx))], columns=data.columns)
```

```
            xx_label = pd.Series(xx_label)
            xx_label.name = 'output'

            return xx, xx_label
```

Let's split the data to train and test again,

```
In [15]: merged = pd.concat([train_num, test_num])
         merged['subject'] = pd.concat([train['subject'], test['subject']])

         group_means = merged.groupby('subject').mean()

         group_corr = group_means.transpose().corr()
         np.fill_diagonal(group_corr.values, np.nan)
         print('Most similar subjects:')
         holdouts = group_corr[['E','J']].idxmax()
         print(holdouts)
```

```
Most similar subjects:
subject
E    A
J    M
dtype: object
```

The `A` and `M` subjec are being hold for the second validation set.

```
In [16]: holdout_subjects = np.zeros((len(train_all)), dtype=bool)
         for item in holdouts:
             colName = "subject_"+item
             for rowIdx in range(len(train_all)):
                 if train_all[colName][rowIdx]:
                     holdout_subjects[rowIdx] = True

         holdout_subjects = pd.Series(holdout_subjects)

         frac = 0.25

         # validation set with all subjects
         X_train1, X_test1, y_train1, y_test1 = train_test_split(train_all, train_labels, test_

         # validation set without subjects A or M
         frac2 = (frac - np.mean(holdout_subjects)) * len(holdout_subjects) / np.sum(~holdout_s

         X_train2, X_test2, y_train2, y_test2 = train_test_split(train_all.loc[~holdout_subject
                                                                 train_labels.loc[~holdout_subj
                                                                 test_size=frac2, random_state=
         X_test2 = pd.concat([X_test2, train_all.loc[holdout_subjects,:]])
         y_test2 = pd.concat([y_test2, train_labels.loc[holdout_subjects]])
```

```
        # Over-sample training data [percentage = ratio of class 0 to class 1]
        X_train1, y_train1 = overSample(X_train1, y_train1, 12, 1.0)
        X_train2, y_train2 = overSample(X_train2, y_train2, 12, 1.0)

        X_train1_original, X_test1_original = X_train1, X_test1
        X_train2_original, X_test2_original = X_train2, X_test2
```

The following function performs PCA on training and validation dataset. Both training and test datasets are mapped to PCA transformation of the corresponding training dataset.

```
In [17]: # PCA Transform
         def get_PCA(dataTr, dataTe, ncomp=200):
             pca = PCA(n_components=ncomp)
             pca.fit(dataTr)

             # Training
             tmpDF1 = pca.transform(dataTr)
             pcNames = ['PC'+str(i+1) for i in range(tmpDF1.shape[1])]
             tmpDF = pd.DataFrame(tmpDF1, index=[i for i in range(tmpDF1.shape[0])], columns=pc

             # Test
             tmpDF2 = pca.transform(dataTe)
             pcNames = ['PC'+str(i+1) for i in range(tmpDF2.shape[1])]
             tmpDF2 = pd.DataFrame(tmpDF2, index=[i for i in range(tmpDF2.shape[0])], columns=p
             return tmpDF1, tmpDF2
```

```
In [18]: # Sklearn classifier
         class SklearnHelper:
             def __init__(self, clf, seed=0, params=None):
                 params['random_state'] = seed
                 self.clf = clf(**params)

             def train(self, x_train, y_train):
                 self.clf.fit(x_train, y_train)

             def predict(self, x):
                 return self.clf.predict(x)

             def predict_proba(self, x):
                 return self.clf.predict_proba(x)

             def fit(self,x,y):
                 return self.clf.fit(x,y)

             def feature_importances(self,x,y):
                 print(self.clf.fit(x,y).feature_importances_)
```

```
In [19]: def plotROC(y_test, scores_test, y_train=None, scores_train=None):
             auc = []
```

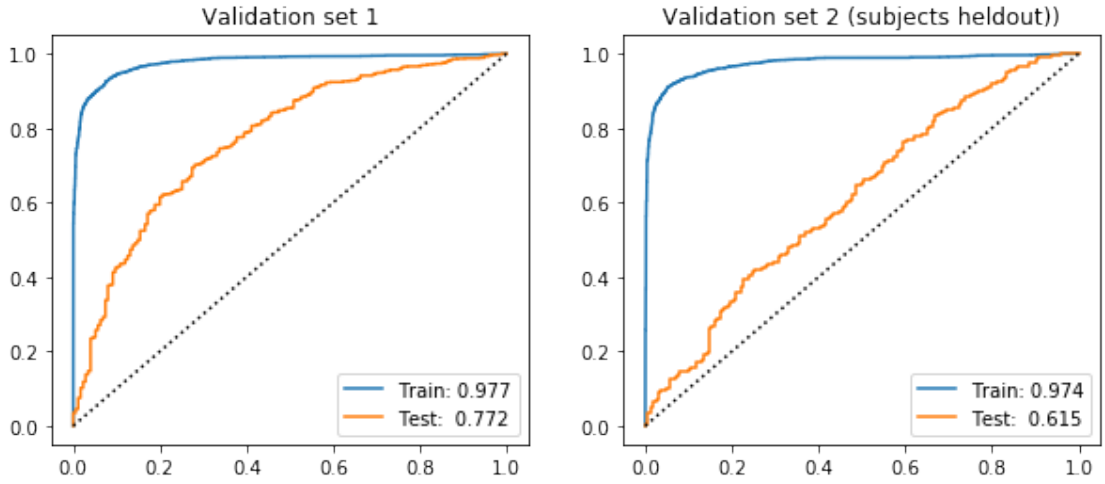
```
    if y_train is not None:
        fpr, tpr, thresholds = roc_curve(y_train, scores_train)
        auc.append(roc_auc_score(y_train, scores_train))
        plt.plot(fpr,tpr,label='Train: %1.3f' % auc[0])

    fpr, tpr, thresholds = roc_curve(y_test, scores_test)
    auc.append(roc_auc_score(y_test, scores_test))
    plt.plot(fpr,tpr,label='Test:  %1.3f' % auc[-1])
    plt.plot([0,1], [0,1], ':k')
    plt.legend(loc='lower right')
    return auc

def predictAll(pipe1, pipe2=None, iPCA=False):

    if not iPCA:
        X_train1, X_test1 = X_train1_original, X_test1_original
        X_train2, X_test2 = X_train2_original, X_test2_original
    if iPCA:
        X_train1, X_test1 = get_PCA(X_train1_original, X_test1_original)
        X_train2, X_test2 = get_PCA(X_train2_original, X_test2_original)

    fit = pipe2 is None

    if fit:
        pipe1.fit(X_train1, y_train1)
    scores_train = pipe1.predict_proba(X_train1)
    scores_test = pipe1.predict_proba(X_test1)

    plt.figure(figsize=(10, 4))
    plt.subplot(1,2,1)
    auc1 = plotROC(y_test1, scores_test[:,1], y_train1, scores_train[:,1])
    plt.title('Validation set 1')

    if fit:
        pipe1.fit(X_train2, y_train2)
        scores_train = pipe1.predict_proba(X_train2)
        scores_test = pipe1.predict_proba(X_test2)
    else:
        scores_train = pipe2.predict_proba(X_train2)
        scores_test = pipe2.predict_proba(X_test2)

    plt.subplot(1,2,2)
    auc2 = plotROC(y_test2, scores_test[:,1], y_train2, scores_train[:,1])
    plt.title('Validation set 2 (subjects heldout))')

    return auc1, auc2
```

```
def submitOutput(pipe0, fileName, iPCA=False):
    fileName += ".csv"
    if iPCA:
        dummy, test_PCA = get_PCA(X_train1_original, test_all)
        scores = pipe0.predict_proba(test_PCA)
        tmp = pd.DataFrame([test_PCA.index, scores[:,1]]).T
    else:
        scores = pipe0.predict_proba(test_all)
        tmp = pd.DataFrame([test_all.index, scores[:,1]]).T

    tmp.columns = ['id','output']
    tmp.id = tmp.id.astype(int)
    tmp.to_csv(fileName, index=False)
    return
```

1.1 Support Vector Machine (SVM)

```
In [20]: estimators_SVM = []
         estimators_SVM.append(('SVM', SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0
                                           decision_function_shape='ovr', degree=3, gamma='auto
                                           max_iter=-1, probability=True, random_state=0, shrin
                                           tol=0.001, verbose=False) ))
         SVM_pipe = Pipeline(estimators_SVM)
         predictAll(SVM_pipe)
```

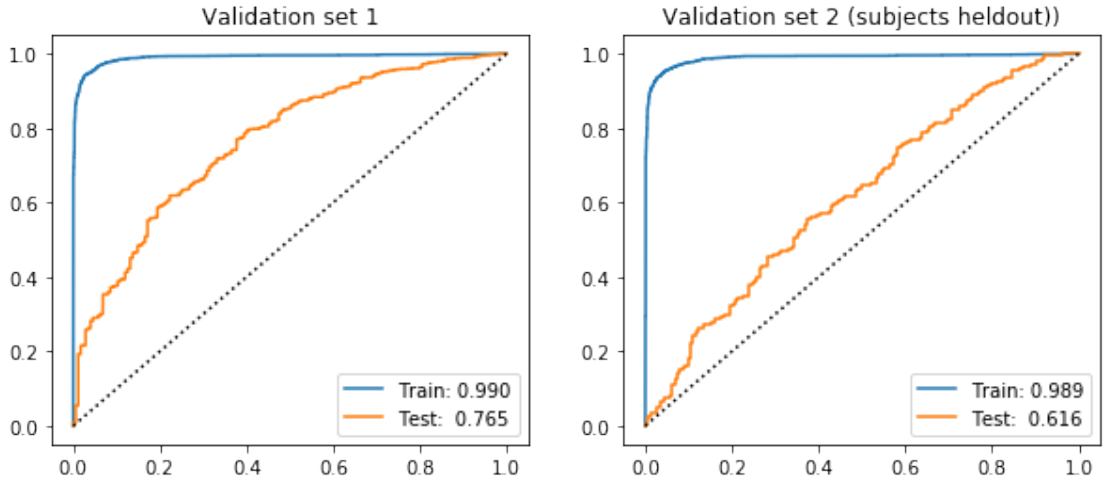
```
Out[20]: ([0.9767490254825566, 0.7716433720759159],
          [0.9740466488168859, 0.6147501122250487])
```



1.1.1 SVM with PCA Transformation

```
In [21]: predictAll(SVM_pipe, iPCA=True)
```

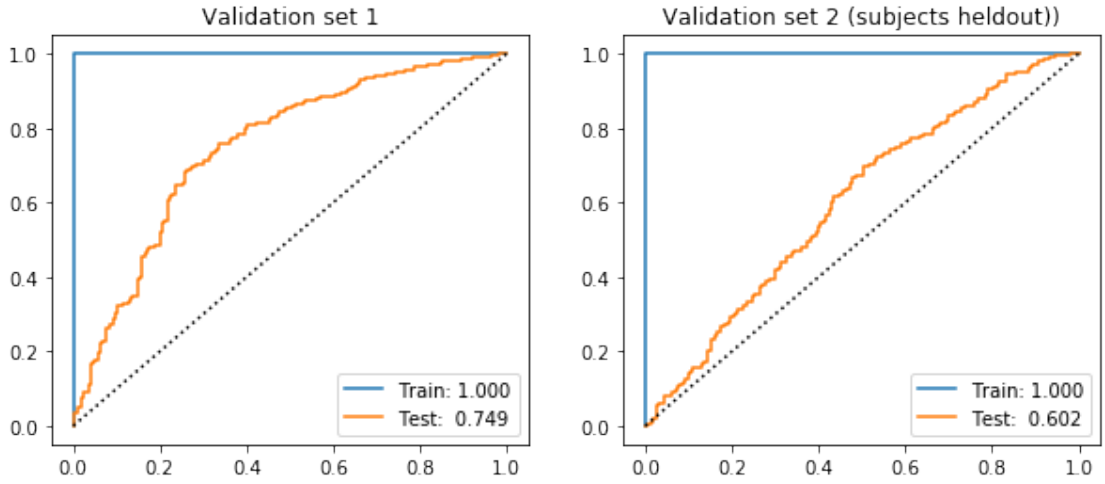
```
Out[21]: ([0.990266046705592, 0.7648903928203619],
          [0.9886319373529966, 0.615685320963639])
```



1.2 Nu-Support Vector Classification (NuSVC)

```
In [22]: estimators_NuSVC = []
         estimators_NuSVC.append(('NuSVC', NuSVC(nu=0.1, kernel='rbf', degree=3, probability=Tr


         NuSVC_pipe = Pipeline(estimators_NuSVC)
         predictAll(NuSVC_pipe)
```

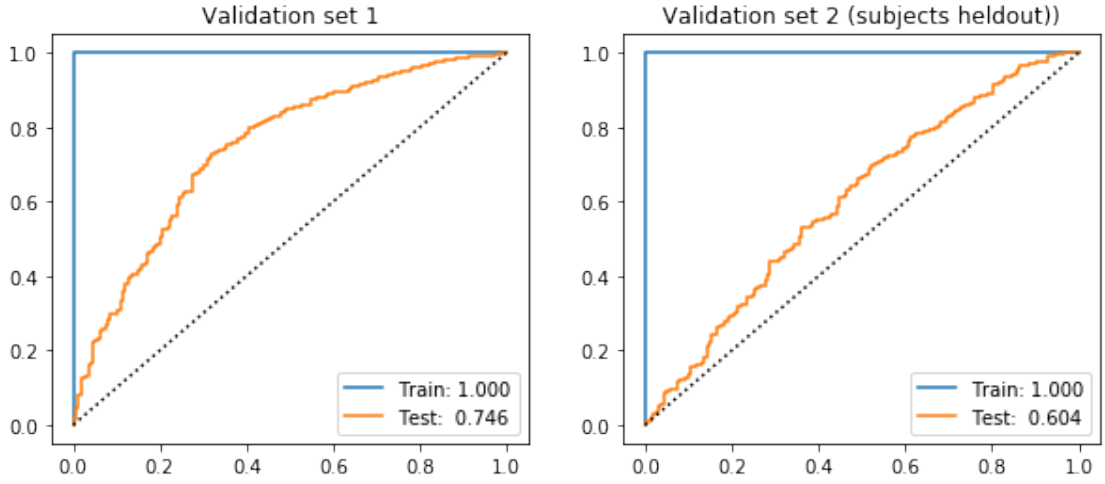
```
Out[22]: ([0.9999039375396895, 0.7491128439017214],
          [0.9997462354970532, 0.6023991221507307])
```



1.2.1 NuSVC with PCA Transformation

```
In [23]: predictAll(NuSVC_pipe, iPCA=True)
```

```
Out[23]: ([0.999999643773324, 0.7458584669707224],
          [0.9999688485501577, 0.6037333532844531])
```

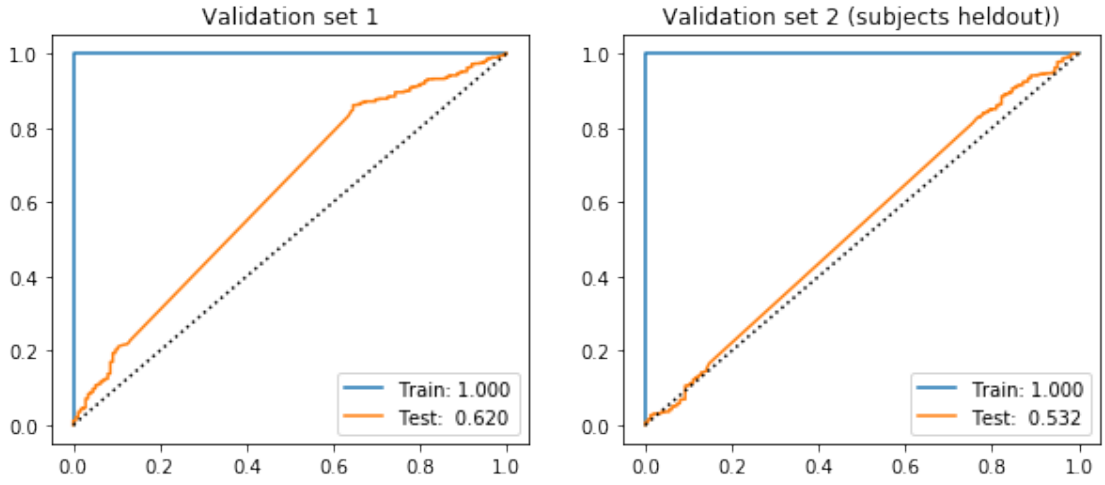



```
In [24]: #submitOutput(NuSVC_pipe, "outputSVM_4")
```

1.3 Gaussian Process

```
In [25]: estimators_GP = []
         estimators_GP.append(('GaussianProcessClassifier', GaussianProcessClassifier(max_iter
                                                                                       random_s

         GP_pipe = Pipeline(estimators_GP)
         predictAll(GP_pipe)
```

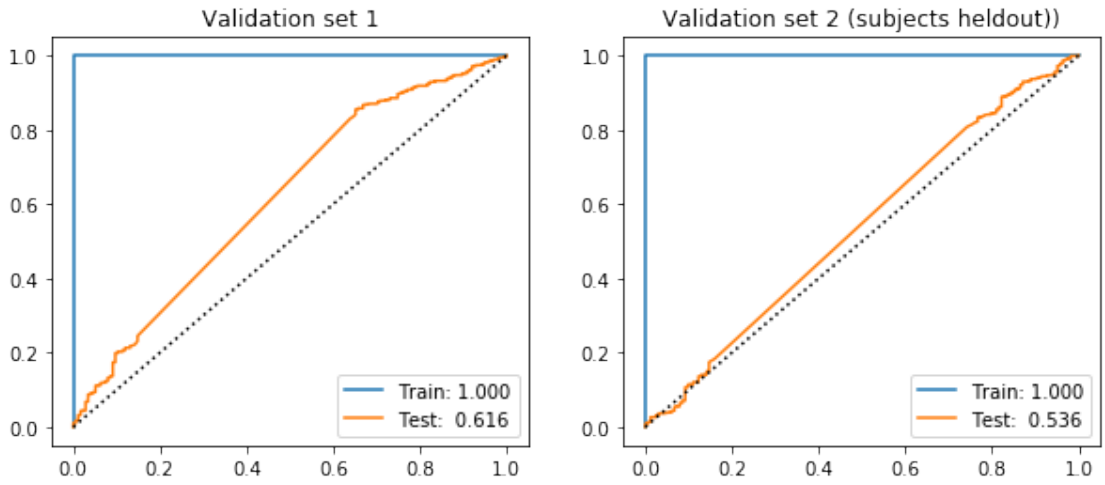
```
Out[25]: ([1.0, 0.6199529204060615], [1.0, 0.5323582223552298])
```



1.3.1 Gaussian Process with PCA Transformation

```
In [26]: predictAll(GP_pipe, iPCA=True)
```

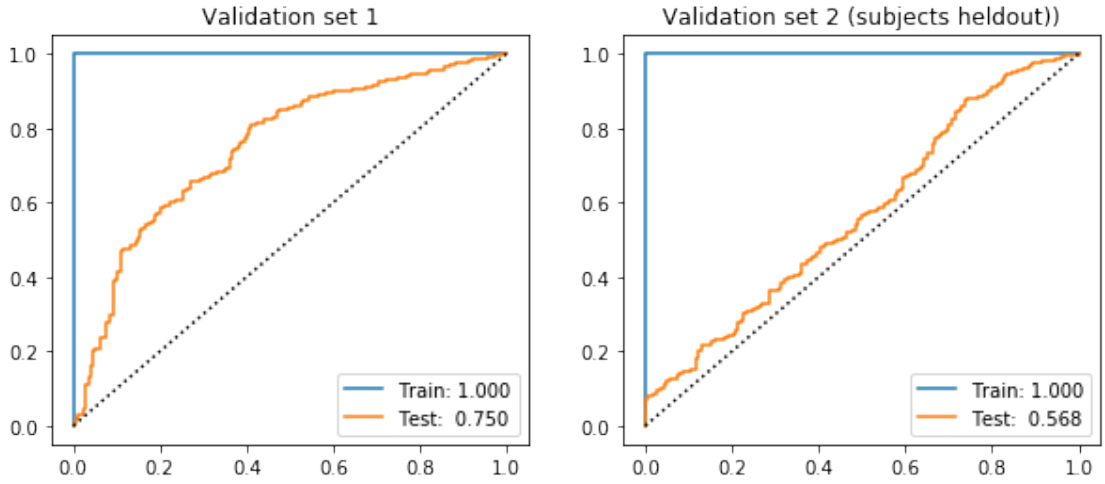
```
Out[26]: ([1.0, 0.6162100926879506], [1.0, 0.5360616489600479])
```



1.4 AdaBoost Classifier

```
In [27]: estimators_ada = []
         estimators_ada.append(('AdaBoostClassifier', AdaBoostClassifier(base_estimator=None, r
                                                                         learning_rate=1.0, alg
                                                                         random_state=0) ))

         ada_pipe = Pipeline(estimators_ada)
         predictAll(ada_pipe)
```

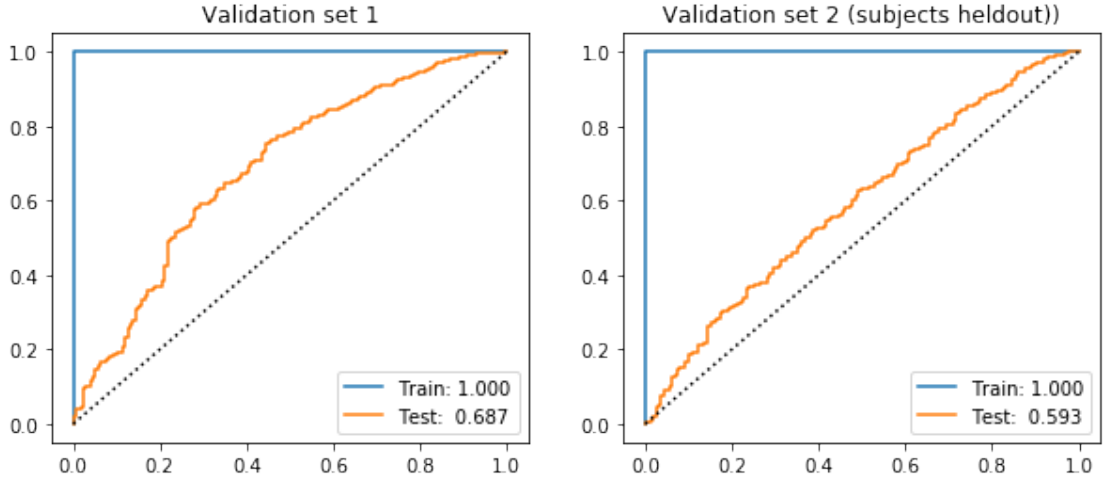
```
Out[27]: ([1.0, 0.7500426658820067], [1.0, 0.5675470098259265])
```



1.4.1 AdaBoost Classifier with PCA Transformation

```
In [28]: predictAll(ada_pipe, iPCA=True)
```

```
Out[28]: ([1.0, 0.6872090628218331], [1.0, 0.5925170831462916])
```

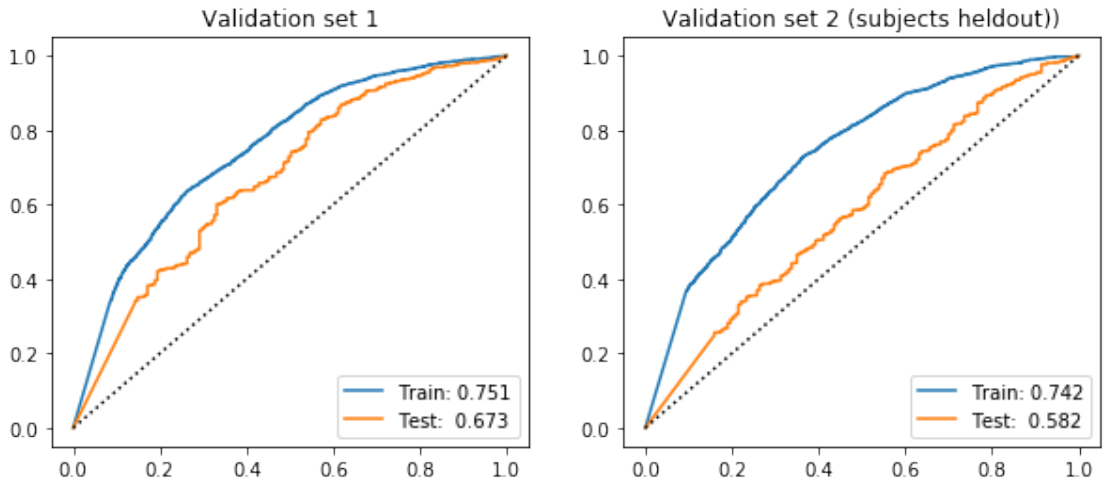



```
In [29]: #submitOutput(ada_pipe, "outputAB_8")
```

1.5 Naive Bayes Classifier

```
In [30]: estimators_NB = []
         estimators_NB.append(('GaussianNB', GaussianNB(priors=None) ))

         NB_pipe = Pipeline(estimators_NB)
         predictAll(NB_pipe)
```

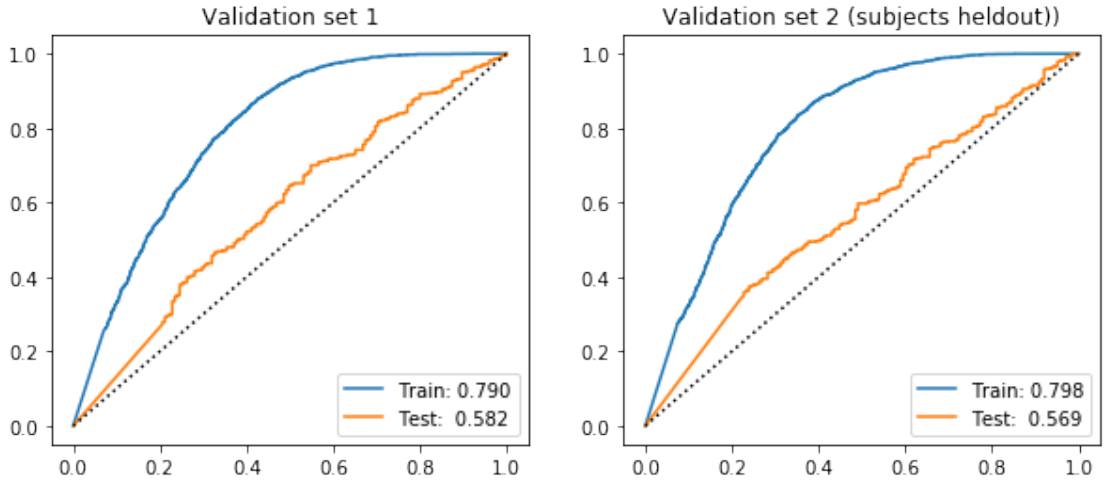
```
Out[30]: ([0.7508445540778217, 0.6730116227747536],
          [0.7424900144626597, 0.5820489799990024])
```



1.5.1 Naive Bayes Classifier with PCA Transformation

```
In [31]: predictAll(NB_pipe, iPCA=True)
```

```
Out[31]: ([0.7895213904619596, 0.5818743563336767],
          [0.7979466641649656, 0.5686599082248491])
```



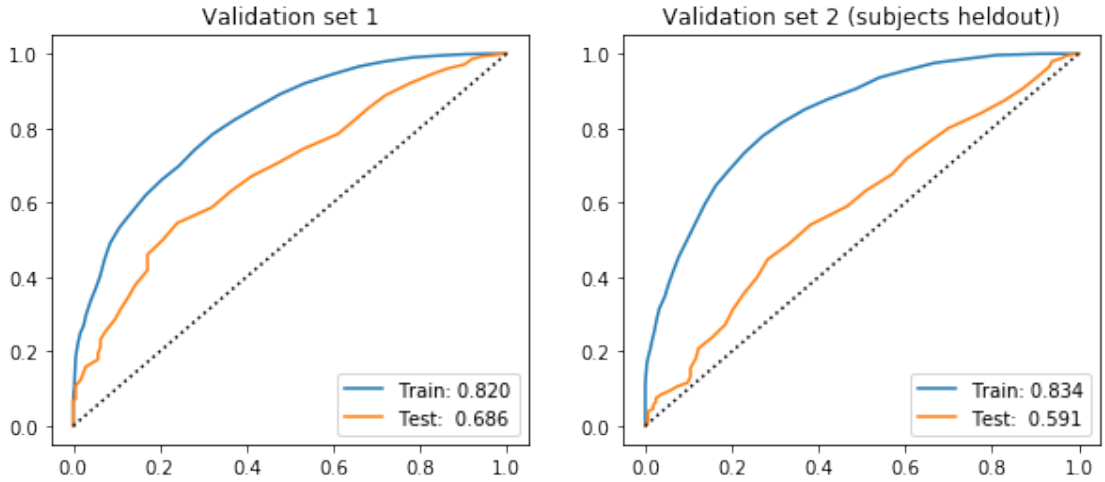
1.6 KN Classifier

```
In [32]: estimators_KN = []
         estimators_KN.append(('KN', KNeighborsClassifier(n_neighbors=50, weights='uniform', al
```

```
                                                    leaf_size=30, p=2, metric='minkowski
```

```
KN_pipe = Pipeline(estimators_KN)
predictAll(KN_pipe)
```

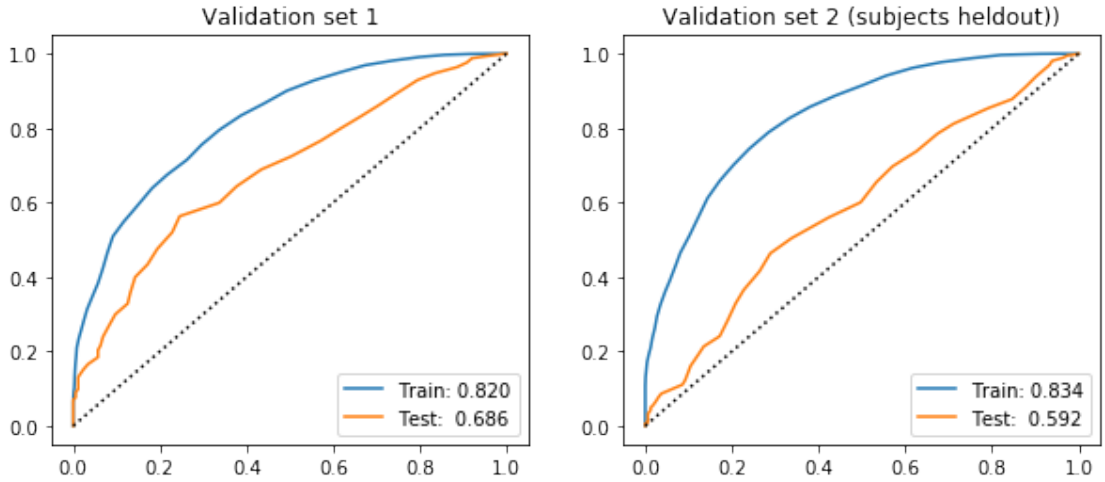
```
Out[32]: ([0.8203161179271787, 0.6862527585699573],
          [0.8341854452997913, 0.5912514339867325])
```



1.6.1 KN Classifier with PCA Transformation

```
In [33]: predictAll(KN_pipe, iPCA=True)
```

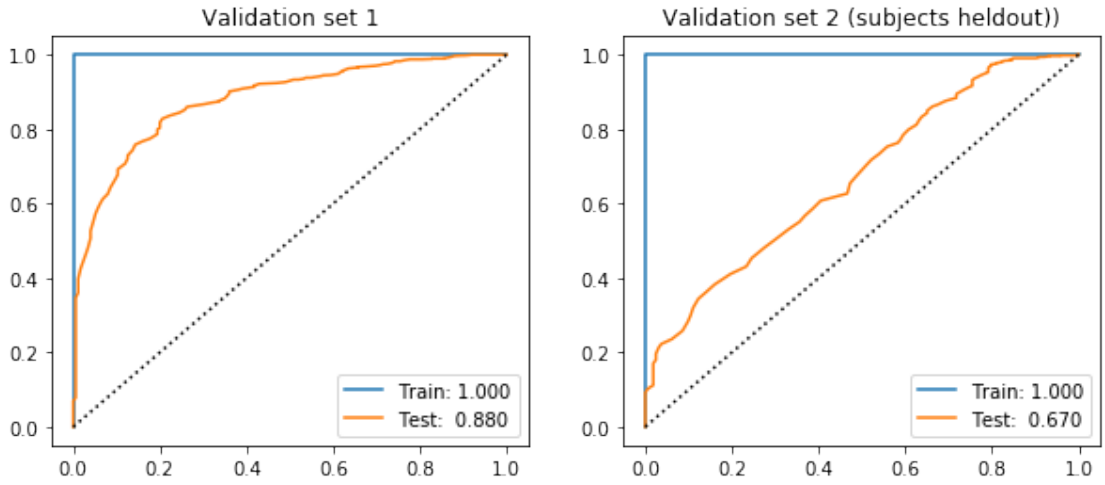
```
Out[33]: ([0.819914828576599, 0.6863910548771517],
          [0.8343987129179431, 0.5923051024988778])
```



1.7 Extra Trees Classifier

```
In [41]: estimators_ET = []
         estimators_ET.append(('ExtraTreesClassifier', ExtraTreesClassifier(n_estimators=200,
                                                                            max_depth=None, mi
                                                                            min_samples_leaf=1
                                                                            max_features='auto
                                                                            min_impurity_decre
                                                                            min_impurity_split
                                                                            oob_score=False, r

         ET_pipe = Pipeline(estimators_ET)
         predictAll(ET_pipe)
```

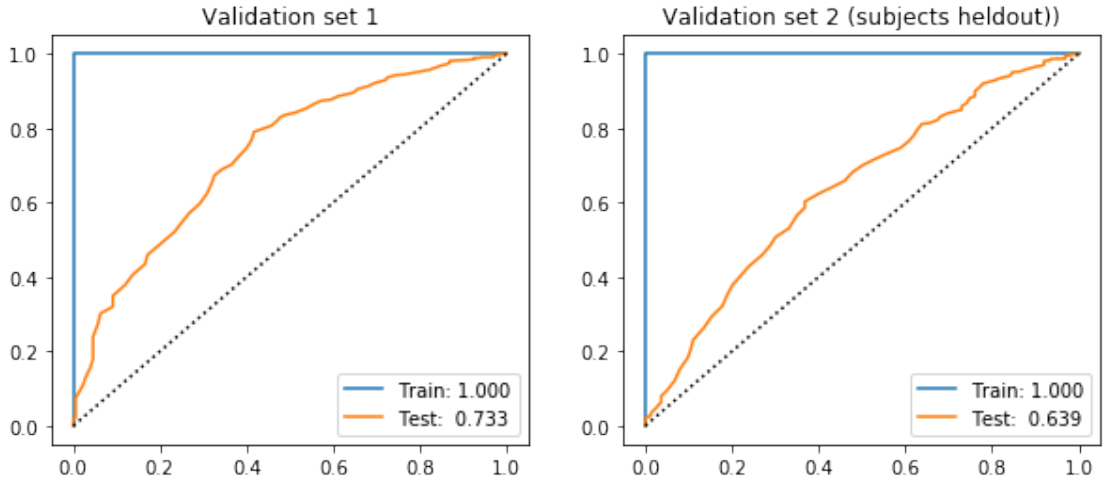
```
Out[41]: ([1.0, 0.8800235397969692], [1.0, 0.6702079904234626])
```



1.7.1 Extra Trees Classifier on PCA Transformed Data

```
In [42]: predictAll(ET_pipe, iPCA=True)
```

```
Out[42]: ([1.0, 0.7334765337648963], [1.0, 0.639102947777944])
```

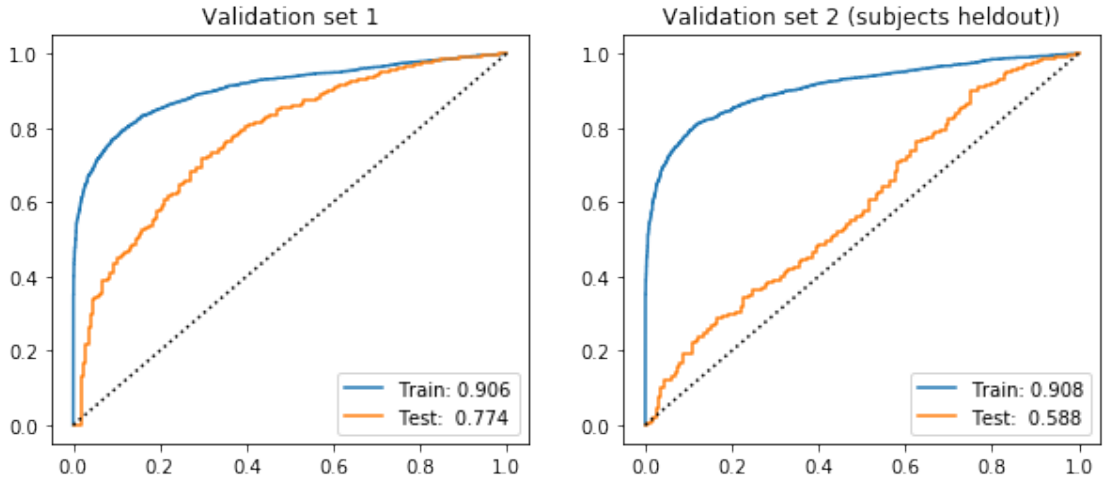


In [36]: *#submitOutput(ET_pipe, "outputET_7")*

1.8 Logistic Regression Classifier

In [37]:
```
estimators_LG = []
estimators_LG.append(('LogisticRegression',  LogisticRegression(penalty='l2', dual=Fa
                                                                fit_intercept=True, i
                                                                class_weight=None, ra
                                                                max_iter=1000)))


LG_pipe = Pipeline(estimators_LG)
predictAll(LG_pipe)
```

Out[37]:
```
([0.9058064235744163, 0.7738943651611004],
 [0.9076712961315531, 0.5882899396478628])
```



1.8.1 Logistic Regression Classifier with PCA Transformation

```
In [38]: predictAll(LG_pipe, iPCA=True)
```

```
Out[38]: ([0.8851136909310863, 0.7698160953361777],
          [0.8946525061760744, 0.6184722430046385])
```