homework11

November 24, 2019

NAME: FULLNAME
SECTION: NUMBER
CS 5970: Machine Learning Practices

1 Homework 11: Dimensionality Reduction

1.1 Assignment Overview

Follow the TODOs and read through and understand any provided code.
For all plots, make sure all necessary axes and curves are clearly and accurately labeled. Include figure/plot titles appropriately as well. Post any questions you have to the Canvas discussion.

1.1.1 Task

For this assignment you will be exploring dimensionality reduction using Prinicipal Componenet Analysis (PCA). Having a large number of features can dramatically increase training times and the likelihood of overfitting. Additionally, it's difficult to visualize and understand patterns in high dimensional spaces. It's not uncommon that a lower dimensional subspace of the full feature space will better characterize trends within the data. PCA is one such technique that attempts to locate such subspaces and projects the data into the determined subspace.

1.1.2 Data set

The BMI data will be utilized. Recall:
* *MI* files contain data with the number of activations for 48 neurons, at mutliple time points, for a single fold. There are 20 folds (20 files), where each fold consists of over 1000 times points (the rows). At each time point, we record the number of activations for each neuron for 20 bins. Therefore, each time point has 48 * 20 = 960 columns.
* *theta* files record the angular position of the shoulder (in column 0) and the elbow (in column 1) for each time point.
* *dtheta* files record the angular velocity of the shoulder (in column 0) and the elbow (in column 1) for each time point.
* *torque* files record the torque of the shoulder (in column 0) and the elbow (in column 1) for each time point.
* *time* files record the actual time stamp of each time point.

1.1.3 Objectives

- Dimensionality Reduction
- Principal Component Analysis (PCA)

1.1.4 Notes

- Do not save work within the ml_practices folder

1.1.5 General References

- Guide to Jupyter
- Python Built-in Functions
- Python Data Structures
- Numpy Reference
- Numpy Cheat Sheet
- Summary of matplotlib
- DataCamp: Matplotlib
- Pandas DataFrames
- Sci-kit Learn Linear Models
- Sci-kit Learn Ensemble Models
- Sci-kit Learn Metrics
- Sci-kit Learn Model Selection
- Sci-kit Learn Pipelines
- Sci-kit Learn Preprocessing
- SciPy Paired t-test for Dependent Samples

[2]:
```
import sys

# THESE 3 IMPORTS ARE CUSTOM .py FILES AND CAN BE FOUND
# ON THE SERVER AND GIT
import visualize
import metrics_plots
from pipeline_components import DataSampleDropper, DataFrameSelector
from pipeline_components import DataScaler, DataLabelEncoder

from KFoldHolisticCrossValidation import KFoldHolisticCrossValidation,␣
 ↪generate_paramsets


import pandas as pd
import numpy as np
import seaborn as sns
import scipy.stats as stats
import os, re, fnmatch
import pathlib, itertools
```

```
import time as timelib
import matplotlib.pyplot as plt
import matplotlib.patheffects as peffects

from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import explained_variance_score, confusion_matrix
from sklearn.metrics import mean_squared_error, roc_curve, auc, f1_score
from sklearn.linear_model import LinearRegression, SGDClassifier
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.decomposition import PCA
from sklearn.externals import joblib

FIGWIDTH = 5
FIGHEIGHT = 5
FONTSIZE = 10

plt.rcParams['figure.figsize'] = (FIGWIDTH, FIGHEIGHT)
plt.rcParams['font.size'] = FONTSIZE

plt.rcParams['xtick.labelsize'] = FONTSIZE
plt.rcParams['ytick.labelsize'] = FONTSIZE

%matplotlib inline
#https://matplotlib.org/3.1.1/tutorials/introductory/images.html
plt.style.use('ggplot')
```

[3]:
```
""" PROVIDED
Display current working directory of this notebook. If you are using
relative paths for your data, then it needs to be relative to the CWD.
"""
HOME_DIR = pathlib.Path.home()
pathlib.Path.cwd()
```

[3]: PosixPath('/home/jovyan')

2 LOAD DATA

[4]:
```
""" PROVIDED """
def read_bmi_file_set(directory, filebase):
    '''
    Read a set of CSV files and append them together
    :param directory: The directory in which to scan for the CSV files
    :param filebase: A file specification that potentially includes wildcards
    :returns: A list of Numpy arrays (one for each fold)
    '''

    # The set of files in the directory
    files = fnmatch.filter(os.listdir(directory), filebase)
    files.sort()

    # Create a list of Pandas objects; each from a file in the directory that␣
 ↪matches filebase
    lst = [pd.read_csv(directory + "/" + file, delim_whitespace=True).values␣
 ↪for file in files]

    # Concatenate the Pandas objects together.  ignore_index is critical here␣
 ↪so that
    # the duplicate row indices are addressed
    return lst
```

[5]:
```
""" TODO
Load the BMI data from all the folds
From the MI_folds we will predict torque_folds
"""
# TODO: set path appropriately
dir_name = str(HOME_DIR / 'ml_practices/imports/datasets/bmi/DAT6_08')
MI_folds = read_bmi_file_set(dir_name, 'MI_fold*')
theta_folds = read_bmi_file_set(dir_name, 'theta_fold*')
dtheta_folds = read_bmi_file_set(dir_name, 'dtheta_fold*')
torque_folds = read_bmi_file_set(dir_name, 'torque_fold*')
time_folds = read_bmi_file_set(dir_name, 'time_fold*')

nfolds = len(MI_folds)
nfolds
```

[5]: 20

[6]:
```
""" PROVIDED
Print out the shape of all the data for each fold
"""
# Zip all data together for convenience when looping
```

```
alldata_folds = zip(MI_folds, theta_folds, dtheta_folds,
                    torque_folds, time_folds)
for i, (MI, theta, dtheta, torque, time) in enumerate(alldata_folds):
    print("FOLD %2d " % i, MI.shape, theta.shape,
          dtheta.shape, torque.shape, time.shape)
```

```
FOLD  0  (1193, 960) (1193, 2) (1193, 2) (1193, 2) (1193, 1)
FOLD  1  (1104, 960) (1104, 2) (1104, 2) (1104, 2) (1104, 1)
FOLD  2  (1531, 960) (1531, 2) (1531, 2) (1531, 2) (1531, 1)
FOLD  3  (1265, 960) (1265, 2) (1265, 2) (1265, 2) (1265, 1)
FOLD  4  (1498, 960) (1498, 2) (1498, 2) (1498, 2) (1498, 1)
FOLD  5  (1252, 960) (1252, 2) (1252, 2) (1252, 2) (1252, 1)
FOLD  6  (1375, 960) (1375, 2) (1375, 2) (1375, 2) (1375, 1)
FOLD  7  (1130, 960) (1130, 2) (1130, 2) (1130, 2) (1130, 1)
FOLD  8  (1247, 960) (1247, 2) (1247, 2) (1247, 2) (1247, 1)
FOLD  9  (1257, 960) (1257, 2) (1257, 2) (1257, 2) (1257, 1)
FOLD 10  (1265, 960) (1265, 2) (1265, 2) (1265, 2) (1265, 1)
FOLD 11  (1146, 960) (1146, 2) (1146, 2) (1146, 2) (1146, 1)
FOLD 12  (1225, 960) (1225, 2) (1225, 2) (1225, 2) (1225, 1)
FOLD 13  (1238, 960) (1238, 2) (1238, 2) (1238, 2) (1238, 1)
FOLD 14  (1570, 960) (1570, 2) (1570, 2) (1570, 2) (1570, 1)
FOLD 15  (1359, 960) (1359, 2) (1359, 2) (1359, 2) (1359, 1)
FOLD 16  (1579, 960) (1579, 2) (1579, 2) (1579, 2) (1579, 1)
FOLD 17  (1364, 960) (1364, 2) (1364, 2) (1364, 2) (1364, 1)
FOLD 18  (1389, 960) (1389, 2) (1389, 2) (1389, 2) (1389, 1)
FOLD 19  (1289, 960) (1289, 2) (1289, 2) (1289, 2) (1289, 1)
```

[7]:
```
""" PROVIDED
Summary statistics
"""
print("Means")
all_MI = np.concatenate(MI_folds, axis=0)
all_theta = np.concatenate(theta_folds, axis=0)
all_dtheta = np.concatenate(dtheta_folds, axis=0)
all_torque = np.concatenate(torque_folds, axis=0)
all_time = np.concatenate(time_folds, axis=0)

df = np.concatenate(([all_MI.mean()], np.mean(all_theta, axis=0), np.
 ↪mean(all_dtheta, axis=0),
                     np.mean(all_torque, axis=0))).reshape(1,-1)
df = pd.DataFrame(df, columns=['MI', 'Should. angle', 'Elbow angle',
                               'Should. d_angle', 'Elbow d_angle',
                               'Should. torque', 'Elbow torque'])
df
```

```
Means
```

[7]:
```
          MI  Should. angle  Elbow angle  Should. d_angle  Elbow d_angle  \
0  0.521546       0.178226     1.617756          0.01136      -0.005352

   Should. torque  Elbow torque
0       -0.000568      0.001702
```

3 REGRESSION

From the MI_folds we will predict torque_folds

[8]:
```
""" PROVIDED
Evaluate the training performance of an already trained model
"""
def compute_rmse(x, y):
    return np.sqrt(np.nanmean((x - y)**2))

def predict_score_rmse(model, X, y):
    '''
    Compute the model predictions and cooresponding scores.
    PARAMS:
        X: feature data
        y: corresponding output
    RETURNS:
        rmse: root mean squared error
        score: score computed by the models score() method
        preds: predictions of the model from X
    '''
    preds = model.predict(X)
    score = model.score(X, y)
    rmse = compute_rmse(y, preds)
    return rmse, score, preds

def predict_plot(model, X, y, time, titles, xlims=None):
    '''
    Compute the model's predicted output
    PARAMS:
        model: already trained model
        X: inputs
        y: outputs
        * For plots
        time: time axis of timestamps
        titles: subplot titles for each output column
        xlims: two element list of the x limits for the plot
    '''
    # Compute and evaulate  predictions on the model
    rmse, score, preds = predict_score_rmse(model, X, y)
```

```
    print("RMSE: %.3f" % rmse)
    print("R^2: %.3f" % score)

    noutputs = y.shape[1]

    # Construct the plots
    fig, axs = plt.subplots(noutputs,1, figsize=(25,4))
    fig.subplots_adjust(hspace=.5)
    axs = axs.ravel()
    for i, ax in enumerate(axs):
        ax.plot(time, preds[:,i], 'r', label='Prediction')
        ax.plot(time, y[:,i], 'b', label='True')
        ax.set(title=titles[i], ylabel=r'$\tau$ (N/m)')
        ax.set(xlim=xlims)
    axs[-1].set(xlabel='Time (s)')
    axs[0].legend()
```

[9]:
```
""" TODO
Obtain the first 1 folds (i.e. index 0)

Split the data into X (i.e. the MI_folds) and y (i.e. the torque_folds).

Hold out a subset of the data, before training and cross validation
"""
# List of the output cloumn names
output_names = ['Shoulder', 'Elbow']

# TODO: Grab the first fold
Xtrain = MI_folds[0]
ytrain = torque_folds[0]
time_trn = time_folds[0]

# TODO: Obtain 2nd to last fold for validation
Xval = np.concatenate(MI_folds[1:-1], axis=0)# TODO
yval = np.concatenate(torque_folds[1:-1], axis=0)# TODO
time_val = np.concatenate(time_folds[1:-1], axis=0)# TODO

# TODO: Obtain last fold for testing
Xtest = MI_folds[-1]# TODO
ytest = torque_folds[-1]# TODO
time_test = time_folds[-1]# TODO

nfeatures = Xtrain.shape[1]

Xtrain.shape, ytrain.shape, Xval.shape, yval.shape, Xtest.shape, ytest.shape
```

[9]:
```
((1193, 960), (1193, 2), (23794, 960), (23794, 2), (1289, 960), (1289, 2))
```

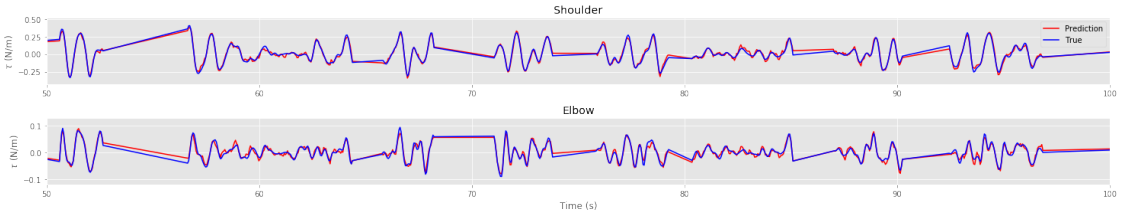
4 BENCHMARK

The task is to predict shoulder and elbow **torque** from the neural activations. We are going to compare the performance of the LinearRegression model trained on the original data to the LinearRegression model trained on the PCA transformed data.

4.1 LinearRegresson Benchmark

[10]:
```
""" PROVIDED
LinearRegression benchmark for comparision
"""
benchmark_lnr = LinearRegression()
benchmark_lnr.fit(Xtrain, ytrain)

# Compute predictions on fully trained model for train set
predict_plot(benchmark_lnr, Xtrain, ytrain, time_trn,
             output_names, xlims=[50,100])
```

```
RMSE: 0.017
R^2: 0.976
```



[11]:
```
# Compute predictions on fully trained model for val set
predict_plot(benchmark_lnr, Xval, yval, time_val,
             output_names, xlims=[2675,2725])
```

```
RMSE: 0.108
R^2: 0.297
```

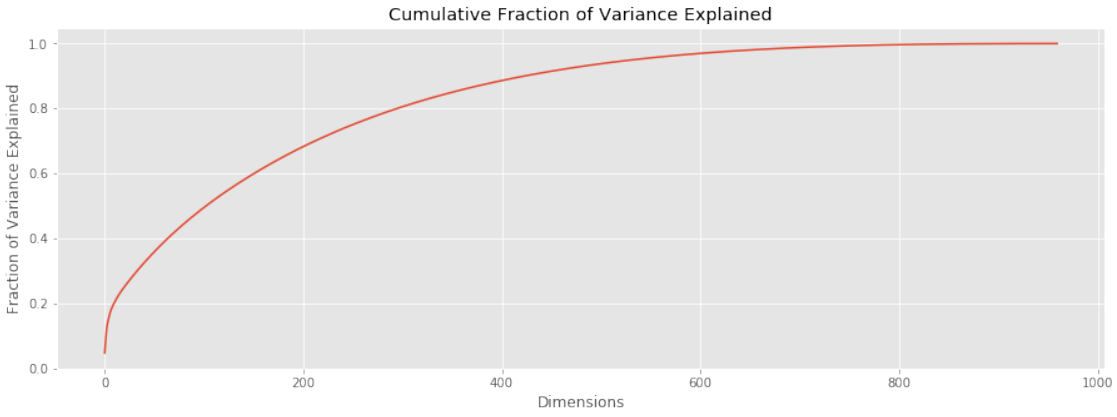
5 Principal Component Analysis

[12]:
```
""" TODO
Create a PCA object and fit it on the training set with whiten=True
"""
pca= PCA( whiten=True)
pca.fit(Xtrain)
```

[12]:
```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=True)
```

[13]:
```
""" TODO
Get an idea of the number of PCs neccessary to represent the data
Use pca.explained_variance_ratio to get a fraction for each
corresponding PC, and np.cumsum() to get the cumuluative sums as
each component is successively considered.
"""
# TODO: Compute the cumulative fraction of explained variance
explained = np.cumsum(pca.explained_variance_ratio_)# TODO

# Plot the cumulative fraction of explained variance
plt.figure(figsize=(FIGWIDTH*3,FIGHEIGHT))
plt.plot(explained)
plt.xlabel('Dimensions')
plt.ylabel('Fraction of Variance Explained')
plt.title('Cumulative Fraction of Variance Explained')
```

[13]:
```
Text(0.5, 1.0, 'Cumulative Fraction of Variance Explained')
```



[14]:
```
""" TODO
Obtain the minimum number of PCs necessary to account for 95% of
the total variance. You can use np.where to locate the indices in
```

```
the cumulative sum that is greater than or equal to .95, and then
add 1 to the list of indices returned to get the number of PCs.
The first element in the list is the minimum number of PCs to
account for 95% of the variance.
"""
majority_explained = np.where(explained>=.95)[0]+1# TODO
# Display the determined number of PCs
nPCs = majority_explained[0]
nPCs
```

[14]:
```
533
```

[15]:
```
""" TODO
Using the number of PCs obtained above, re-fit the PCA with
whiten=True and project the training data into PC space
"""
pca = PCA(n_components= nPCs, whiten= False)# TODO
pca.fit(Xtrain)

# TODO: Project into PC-space
Xtrain_pca = pca.transform(Xtrain)# TODO
Xtrain_pca.shape
```

[15]:
```
(1193, 533)
```

[16]:
```
# TODO: Project back into the original space
Xtrain_recon = pca.inverse_transform(Xtrain_pca)# TODO
Xtrain_recon.shape
```

[16]:
```
(1193, 960)
```

[17]:
```
# TODO: Compute the reconstruction error
print('RMSE of reconstruction error: %.3f'%compute_rmse(Xtrain, Xtrain_recon))
```

```
RMSE of reconstruction error: 0.151
```

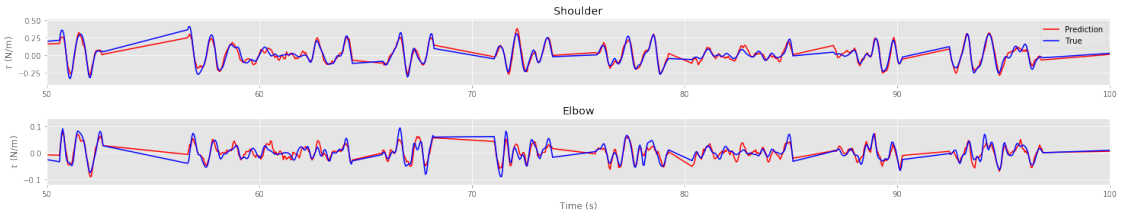
[18]:
```
""" TODO
Implement a model Pipeline. The first step of the pipeline is
PCA with n_components set to the number of PCs determined above
and whiten to true; and the second step of the pipeline is
LinearRegression()
"""
# TODO: Create Pipeline model
pca_model = Pipeline([
    ('PCA', PCA(n_components= nPCs, whiten=True)),
    ('Regression', LinearRegression())
])
```

```
# TODO: Fit model to entire train set
pca_model.fit(Xtrain, ytrain)
```

[18]:
```
Pipeline(memory=None,
     steps=[('PCA', PCA(copy=True, iterated_power='auto', n_components=533,
random_state=None,
  svd_solver='auto', tol=0.0, whiten=True)), ('Regression',
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False))])
```

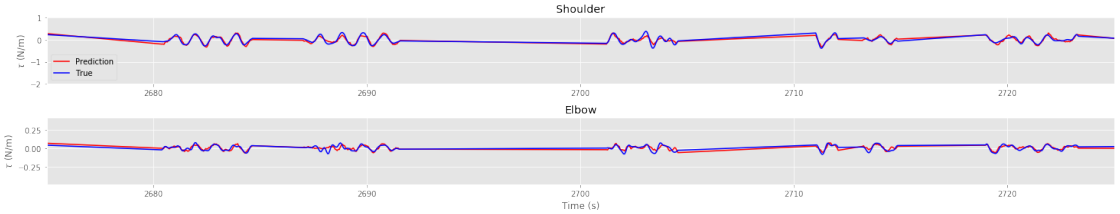
[19]:
```
# TODO: Compute predictions on fully trained model for train set
# Display the plot of the true output overlaying the predicted output
# You can use predict_plot() with xlims=[50,100]
predict_plot(pca_model, Xtrain, ytrain, time_trn,
             output_names, xlims=[50,100])
```

```
RMSE: 0.031
R^2: 0.920
```



[20]:
```
# TODO: Compute predictions on fully trained model for val set
# Display the plot of the true output overlaying the predicted output
# You can use predict_plot() with xlims=[2675,2725]

predict_plot(pca_model, Xval, yval, time_val,
             output_names, xlims=[2675,2725])
```

```
RMSE: 0.070
R^2: 0.701
```



5.0.1 GRIDSEARCH KFoldHolisticCrossValidation

Use the KFoldHolisticCrossValidation from the HW 11 folder to show training and validation set performance as a function of data set size. The hyper-parameter you should vary for PCA is n_components. Briefly discuss and interepret the results of the GridSearch in terms of train size, performance, and variations in the hyper-parameters.

[21]:
```
""" PROVIDED
Evaulation function for KFoldHolisticCrossValidation
"""
def mse_rmse(trues, preds):
    '''
    Compute MSE and rMSE for each column separately.
    '''
    mse = np.sum(np.square(trues - preds), axis=0) / trues.shape[0]
    rmse_rads = np.sqrt(mse)
    rmse_degs = rmse_rads * 180 / np.pi
    return mse, rmse_rads, rmse_degs

def score_eval(model, X, y, preds):
    '''
    Compute the model predictions and corresponding scores, for an
    already trained model.
    PARAMS:
        model: model to predict with
        X: input feature data
        y: true output for X
        preds: predicted output for X
    RETURNS: results as a dictionary of numpy arrays
        mse: mean squared error for each column
        rmse_rads: rMSE in radians
        rmse_deg: rMSE in degrees
        evar: explained variance, best is 1.0
        score: score computed by the models score() method
    '''
    score = model.score(X, y)

    mse, rmse_rads, rmse_degs = mse_rmse(y, preds)
    evar = explained_variance_score(y, preds)

    # Dictionary of numpy arrays. The numpy arrays must
    # be row vectors, where each element is the result
    # for a different output, when using multiple regression.
    # The keys of the dictionary are the name of the performance
    # metric, and the values are the numpy row vectors
    results = {'mse': np.reshape(mse, (1, -1)),
               'rmse_rads': np.reshape(rmse_rads, (1, -1)),
```

```
                'rmse_degs': np.reshape(rmse_degs, (1, -1)),
                'evar': np.reshape(evar, (1, -1)),
                'score': np.reshape(score, (1, -1)),
               }
    return results
```

[22]:
```
# List of number of PCs to try
components = np.append(np.logspace(0, 5, num=6, base=3, dtype=int), nPCs)
components
```

[22]:
```
array([  1,   3,   9,  27,  81, 243, 533])
```

[24]:
```
""" TODO
Create the KFoldHolisticCrossValidation object using the PCA
pipeline model created above
Estimated runtime <140min on mlserver
"""
# Grid Search Parameters
opt_metric = 'rmse_degs'
maximize_opt_metric = False
trainsizes = range(1, 11)
rotation_skip = 1

# TODO: GridSearch pipeline hyper-parameters can be specified
# with '__' separated parameter names
hyperparam_grid = {
    'PCA__n_components': components,
    'PCA__whiten': [True]
}
hyperparams = generate_paramsets(hyperparam_grid)
nhyperparams = len(hyperparams)


# TODO: Save Parameters. Set these appropriately
force = False
write_crossval = True
fullcvfname = "hw11_crossval_%02dparams.pkl" % nhyperparams


if force or (not os.path.exists(fullcvfname)):
    # TODO: Create the cross validation object. use score_eval for the eval_func
    crossval = KFoldHolisticCrossValidation(pca_model, hyperparams, score_eval,
                                            opt_metric, maximize_opt_metric,␣
 ↪trainsizes,
                                            rotation_skip)

    t0 = timelib.time()
```

```
    # TODO: Execute cross validation for all parameters and sizes
    crossval_report= crossval.grid_cross_validation(MI_folds, torque_folds)

    # TODO: Save the cross validation object. Can use joblib.dump()
    if write_crossval:
        joblib.dump(crossval, fullcvfname)

    lapsedTime = timelib.time() - t0
    print(" ** Elapsed Time %.2f min" % (lapsedTime / 60))

else:
    # TODO: Load the cross val object from file. Can use joblib.load()
    crossval = joblib.load(fullcvfname)

crossval.model, crossval.rotation_skip, crossval.trainsizes
```

[24]:
```
(Pipeline(memory=None,
      steps=[('PCA', PCA(copy=True, iterated_power='auto', n_components=533,
random_state=None,
   svd_solver='auto', tol=0.0, whiten=True)), ('Regression',
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
          normalize=False))]), 1, range(1, 11))
```

[25]:
```
""" TODO
Display the lists of the best parameter sets for each size
from the cross validation using get_report_best_params_all_sizes
"""

crossval.get_report_best_params_all_sizes()
```

Best Parameter Sets For Each Train Set Size

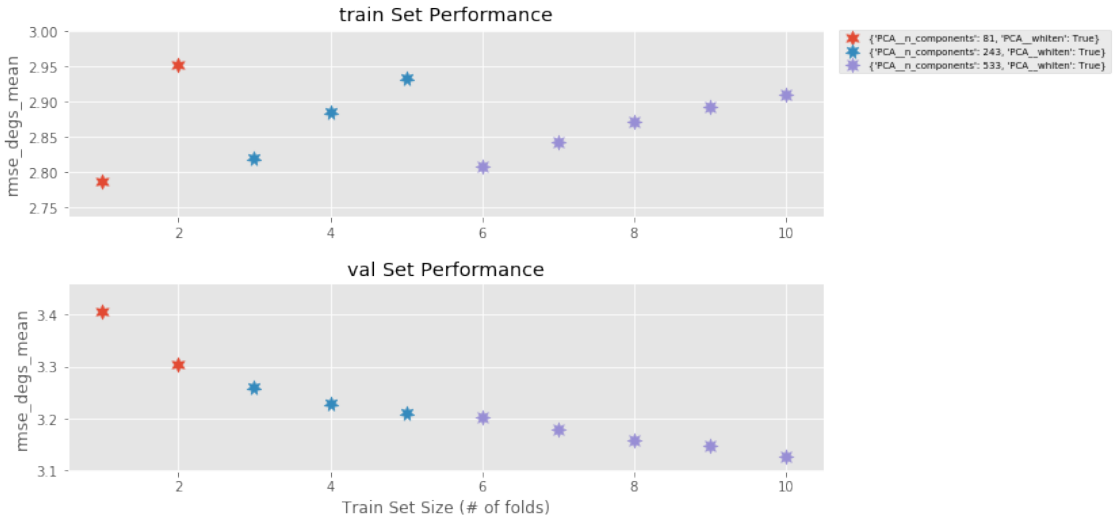
[25]:

	train_size	param_index	paramset
0	1	4	{'PCA__n_components': 81, 'PCA__whiten': True}
1	2	4	{'PCA__n_components': 81, 'PCA__whiten': True}
2	3	5	{'PCA__n_components': 243, 'PCA__whiten': True}
3	4	5	{'PCA__n_components': 243, 'PCA__whiten': True}
4	5	5	{'PCA__n_components': 243, 'PCA__whiten': True}
5	6	6	{'PCA__n_components': 533, 'PCA__whiten': True}
6	7	6	{'PCA__n_components': 533, 'PCA__whiten': True}
7	8	6	{'PCA__n_components': 533, 'PCA__whiten': True}
8	9	6	{'PCA__n_components': 533, 'PCA__whiten': True}
9	10	6	{'PCA__n_components': 533, 'PCA__whiten': True}

[26]:
```
""" TODO
Plot the mean (summary) train and validation set performances for
the best parameter set for each train size for the optimized
```

```
metrics. Use plot_best_params_by_size()
"""
crossval.plot_best_params_by_size()
```

[26]:
```
(<Figure size 720x432 with 2 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f7724baf438>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f7724a92c50>],
       dtype=object))
```



[27]:
```
""" PROVIDED
Display available metrics
"""
crossval.results[0]['results']['val'].keys()
```

[27]:
```
dict_keys(['mse', 'rmse_rads', 'rmse_degs', 'evar', 'score'])
```

[28]:
```
""" PROVIDED
Display available summary (mean and std) metrics
"""
crossval.results[0]['summary']['val'].keys()
```

[28]:
```
dict_keys(['mse_mean', 'mse_std', 'rmse_rads_mean', 'rmse_rads_std',
'rmse_degs_mean', 'rmse_degs_std', 'evar_mean', 'evar_std', 'score_mean',
'score_std'])
```

[29]:
```
""" TODO
Plot the validation results for all parameter sets over all train
sizes, for the specified metrics, rmse_degs_mean and evar_mean
(this variable is declared above). Use plot_allparams_val()
```

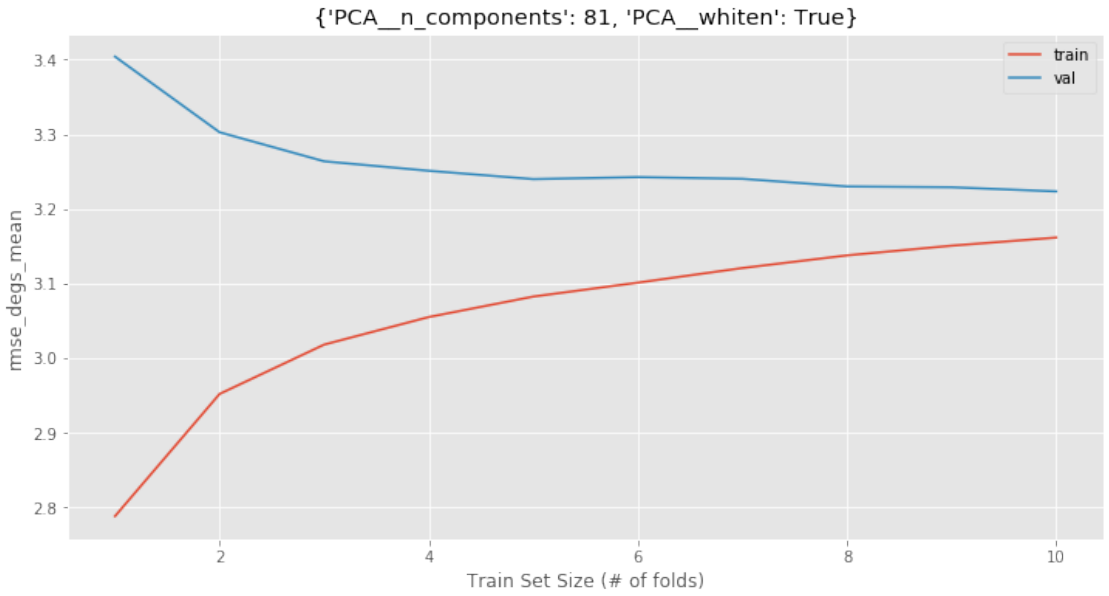
```
"""
metrics = ['rmse_degs_mean', 'evar_mean']

crossval.plot_allparams_val(metrics)
```

[29]:
```
(<Figure size 720x432 with 2 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f772492d048>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f77248e7198>],
       dtype=object))
```

[46]:
```
""" TODO
For the best parameter set for the train set size at
size_idx=0 (i.e. 1 fold), plot just the TRAIN and VAL set performances using
plot_param_train_val() for just the opt_metric
"""
size_idx = 0
print("Train Set Size", trainsizes[size_idx])
bp_idx = crossval.best_param_inds[size_idx]# TODO: obtain the best parameter␣
 ↪index for the size
# TODO: call plot_param_train_val()
crossval.plot_param_train_val([crossval.opt_metric], paramidx= bp_idx, )
```

```
Train Set Size 1
```

[46]:
```
(<Figure size 864x432 with 1 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f77223c6cf8>],
       dtype=object))
```



[34]:
```
""" PROVIDED
Re-fit PCA model with best hyper-parameters for train size of
3 folds
"""
print("Train size %d folds" % trainsizes[0])

bp_idx = crossval.best_param_inds[0]
best_params = crossval.paramsets[bp_idx]

# Set the hyperparameters of the Pipeline model
pca_model.set_params(**best_params)

# Fit the model to entire train set
pca_model.fit(Xtrain, ytrain)
```

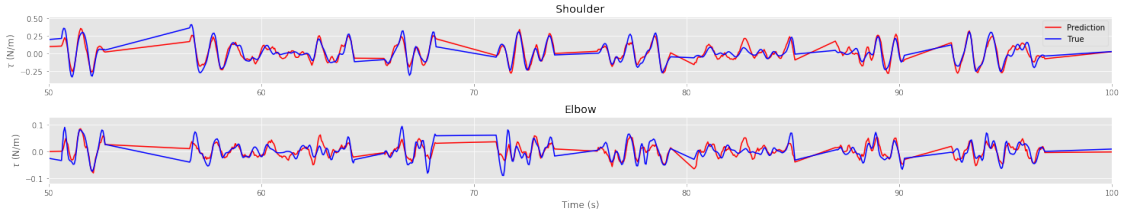
```
Train size 1 folds
```

[34]:
```
Pipeline(memory=None,
     steps=[('PCA', PCA(copy=True, iterated_power='auto', n_components=81,
random_state=None,
  svd_solver='auto', tol=0.0, whiten=True)), ('Regression',
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False))])
```

[37]:
```
# TODO: Compute predictions on fully trained model for train set
# Display the plot of the true output overlaying the predicted output
# You can use predict_plot() with xlims=[50,100]
```

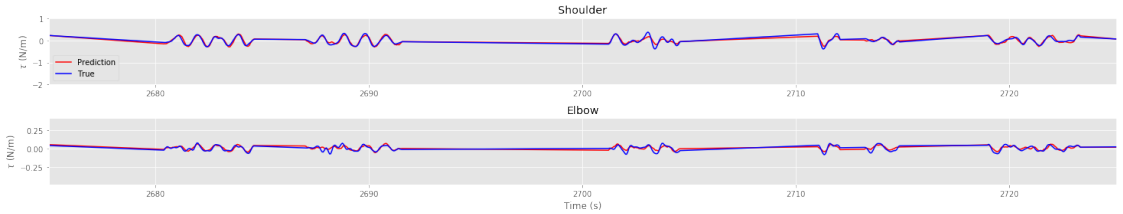
```
predict_plot(pca_model, Xtrain, ytrain, time_trn,
             output_names, xlims=[50,100])
```

```
RMSE: 0.043
R^2: 0.843
```



[38]:
```
# TODO: Compute predictions on fully trained model for val set
# Display the plot of the true output overlaying the predicted output
# You can use predict_plot() with xlims=[2675,2725]

predict_plot(pca_model, Xval, yval, time_val,
             output_names, xlims=[2675,2725])
```

```
RMSE: 0.068
R^2: 0.718
```



6 DISCUSSION

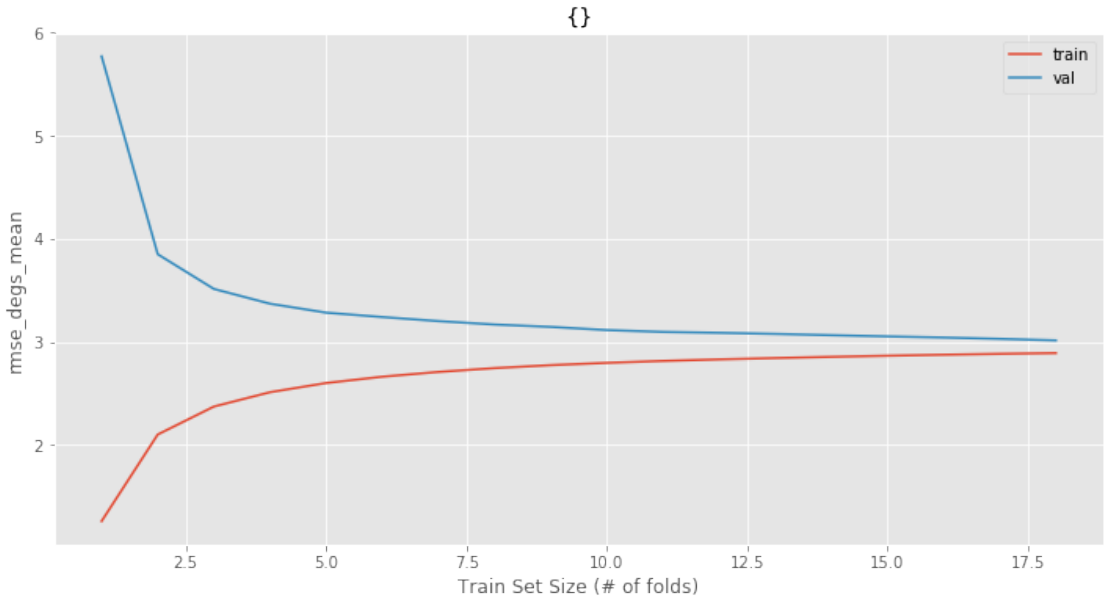
1. Bake off. Compare the training and validation performances of the benchmark linear model learned without PCA to the model learned using PCA for train size of 1 fold. Based on the validation performances, which would you choose and why?
2. Now that you've selected your model, observe and compare the test results. Was your selection justified? Why or why not?

[39]:
```
# TODO: set these paths appropriately
# Re-load saved favorite crossval object
```

```
r_crossval = joblib.load('hw7_full_ridge_crossval.pkl')
# Re-load saved linear crossval object
lnr_crossval = joblib.load('hw7_full_linear_crossval.pkl')
```

[40]:
```
"""
Display Linear Regression model performance
"""
lnr_crossval.plot_param_train_val([lnr_crossval.opt_metric])
```

[40]:
```
(<Figure size 864x432 with 1 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f7722563860>],
       dtype=object))
```

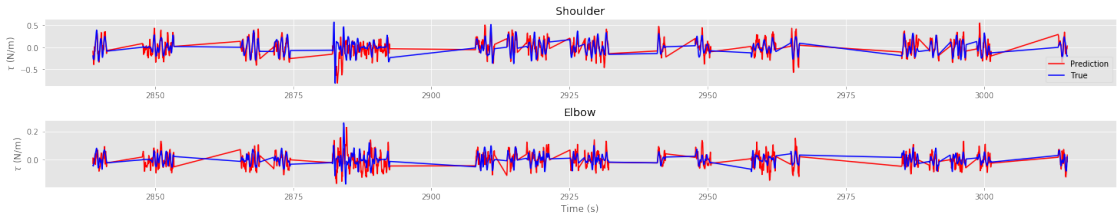


[47]:
```
# TODO: any additional plots or tables relevant to your discussion responses

#TEST sample
predict_plot(pca_model, Xtest, ytest, time_test,
             output_names)
```

```
RMSE: 0.066
R^2: 0.659
```

[48]:
```
predict_plot(benchmark_lnr, Xtest, ytest, time_test,
             output_names)
```

```
RMSE: 0.112
R^2: 0.024
```



1. I will choose PCA model. despite benchmark model obtained RMSE of 0.017 in the training process and grid-search-PCA is higher at 0.043, PCA model did a pretty good job in the validation process with 0.068, close to training process. Conversely, benchmark model got 0.108 which is ten times larger than its own training process. We came to conclusion that it was over-trained. Overall, I would like to choose the model which did better job in validation process.

2. The graphs produced above are the performance of each model (benchmark, PCA) for test samples. Without doubt, PCA is way better than benchmark model from either RMSE or R^2. Hence, we can say our choice is justified by testing samples.