homework6

October 15, 2019

NAME: Zhi Li
SECTION: 113523595
CS 5970: Machine Learning Practices

1 Homework 6: Cross Validation

1.1 Assignment Overview

First read through the entire notebook, do not write any code. This assignment is more complex than previous, and it will be helpful to have a sense of the structure before you start coding.

Follow the TODOs and read through and understand any provided code.
All the plotting functions have been provided. You should not need to alter any of these.

1.1.1 Task

For this assignment you will be implementing **holistic cross validation**. Cross validation is a procedure that involves training, validating, and testing a model on different subsets of the data set to evaluate how well the model will generalize to unseen examples. Additionally, cross validation is a good tool for evaulating models when only small amounts of data are available.

The train sets are utilized for the various models to learn with, the validation sets are utilized to initially evaluate and select the best performing model. The test sets are utilized to determine how well the choosen model actually will generalize to unseen examples.

The validation and test sets can often seem similar conceptually, however, the key difference is that the validation performance is used to actually make guided decisions about model tuning (i.e., hyper-parameter values). Decisions about which hyper-parameters to use are never done based on the test set. The test set performance evaluates the generalized performance on data unused for hyper-parameter selection and training.

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

- Implement and understand **holistic cross validation**
- Training set size sensitivity analysis

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

[1]:
```python
import pandas as pd
import numpy as np
import scipy.stats as stats
import os, re, fnmatch
import pathlib, itertools, time
import matplotlib.pyplot as plt

from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.metrics import explained_variance_score
from sklearn.linear_model import ElasticNet
from sklearn.externals import joblib


FIGW = 10
```

```
FIGH = 6
FONTSIZE = 12

HOME_DIR = pathlib.Path.home()

plt.rcParams['figure.figsize'] = (FIGW, FIGH)
plt.rcParams['font.size'] = FONTSIZE

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

%matplotlib inline
```

[2]:
```
"""
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.
"""
pathlib.Path.cwd()
```

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

2 LOAD DATA

[3]:
```
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
```

[4]:
```python
""" PROVIDED
Load the BMI data from all the folds, using read_bmi_file_set()
"""
# TODO: might need to change; assumes ml_practices is in home directory
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*')

alldata_folds = zip(MI_folds, theta_folds, dtheta_folds,
                    torque_folds, time_folds)

nfolds = len(MI_folds)
nfolds
```

[4]: 20

[5]:
```python
""" PROVIDED
Print out the shape of all the data for each fold
"""
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)
```

3 PARAMETER SET LIST

[6]:
```
""" PROVIDED
Construct the Cartesian product of the parameters
"""
def generate_paramsets(param_lists):
    '''
    Construct the Cartesian product of the parameters
    PARAMS:
        params_lists: dict of lists of values to try for each parameter.
                      keys of the dict are the names of the parameters
                      values are lists of values to try for the
                      corresponding parameter
    RETURNS: a list of dicts that make up the Cartesian product of the
             parameters
    '''
    keys, values = zip(*param_lists.items())
    # Determines cartesian product of parameter values
    combos = itertools.product(*values)
    # Constructs list of dictionaries
    combos_dicts = [dict(zip(keys, vals)) for vals in combos]
    return list(combos_dicts)
```

4 PERFORMANCE EVALUTION

[7]:
```
""" PROVIDED
Evaluate the performance of an already trained model on some data
"""
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

""" TODO
Finish implementation by just returning the dictionary of results
"""
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)
    print(score.shape, mse.shape, rmse_rads.shape, evar.shape)

    # TODO: Complete the results dictionary. This is a
    # 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': rmse_rads.reshape(1,-1),
               'rmse_degs': rmse_degs.reshape(1,-1),
               'evar': np.array([evar]),
               'score': np.array([score]),# TODO
              }
    return results
```

5 CROSS VALIDATION

[8]:
```
""" TODO
Complete KFoldHolisticCrossValidation implementation
General Procedure:
+ iter over hyper-parameter sets
  1. set hyper-parameters of the model
  2. iter over train set sizes
     a. iter over data set splits/rotations
          i. train the model
         ii. evaluate the model on train, val, and test sets
        iii. record the results
     b. record the results by size
  3. record the results by hyper-parameter set
```

```
"""
class KFoldHolisticCrossValidation():
    def __init__(self, model, paramsets, eval_func, opt_metric,
                 maximize_opt_metric=False, trainsizes=[1], rotation_skip=1):
        ''' TODO
        Object for managing and performing cross validation for a given model␣
 ↪for
        a list of parameter sets and train set sizes. Note, train set size is␣
 ↪in
        terms of number of folds (not samples)
        PARAMS:
            model: base ML model

            paramsets: list of dicts of parameter sets to give to the model

            eval_func: function used to evaluate/score the model
                       The eval_func must have the following arguments: model,␣
 ↪X,
                       ytrue, ypreds and return a dict of numpy arrays with␣
 ↪shape
                       1-by-n, where n is the number of outputs if using␣
 ↪multiple
                       regression.
                       template function header: eval_func(model, X, y, preds)
                       template output: {'metrics1':1_by_n_array, ...}

            opt_metric: the optized metric. one of the metric key names␣
 ↪returned
                        from eval_func to use to pick the best parameter sets

            maximize_opt_metric: True if opt_metric is maximized; False if␣
 ↪minimized

            trainsizes: list of training set sizes (in number of folds) to try

            rotation_skip: build model and evaluate every ith rotation (1=all␣
 ↪possible
                           rotations; 2=every other rotation, etc.)
        '''
        # TODO: set the class variables
        self.model = model#TODO
        self.paramsets = paramsets#TODO
        self.trainsizes = trainsizes#TODO
        self.eval_func = eval_func#TODO
        self.opt_metric = opt_metric + '_mean'
        self.maximize_opt_metric = maximize_opt_metric
```

```
        self.rotation_skip = rotation_skip

        # Results attributes
        # Full recording of all results for all paramsets, sizes, rotations,
        # and metrics. This is a list of dictionaries for each paramset
        self.results = None
        # Validation summary report of all means and standard deviations for
        # all metrics, for all paramsets, and sizes. This is a 3D s-by-r-by-p
        # numpy array. Where s is the number of sizes, r the number of summary
        # metrics +2, and p is the number of paramsets
        self.report_by_size = None
        # List of the indices of the best paramset for each size
        self.best_param_inds = None

    def perform_cross_validation(self, all_Xfolds, all_yfolds, trainsize,␣
↪verbose=0):
        ''' TODO: This is where the bulk of the work will be done
        Perform cross validation for a singular train set size and single␣
↪hyper-parameter
        set, by evaluating the model's performance over multiple data set␣
↪rotations all
        of the same size.

        NOTE: This function assumes the hyper-parameters have already been set␣
↪in the model

        PARAMS:
            all_Xfolds: list containing all of the input data folds
            all_yfolds: list containing all of the output data folds
            trainsize: number of folds to use for training
            verbose: flag to display simple debugging information

        RETURNS: train, val, and test set results for all rotations of the data␣
↪sets and
                 the summary (i.e. the averages over all the rotations) of the␣
↪results.
                 results is a dictionary of dictionaries of r-by-n numpy arrays.
↪ Where r
                 is the number of rotations, and n is the number of outputs␣
↪from the model.
                 summary is a dictionary of dictionaries of 1-by-n numpy arrays.
↪

                 General form:
                     results.keys() = ['train', 'val', 'test']
```

```
                results['train'].keys() = ['metric1', 'metric2', ...]

                results['train']['metric1'] = numpy_array

                results =
                {
                    'train':
                               {
                                    'mse'       : r_by_n_numpy_array,
                                    'rmse_rads': r_by_n_numpy_array,
                                    'rmse_degs': r_by_n_numpy_array,
                                    ...
                               },
                    'val'  : {...},
                    'test' : {...}
                }

                summary =
                {
                    'train':
                               {
                                    'mse_mean'       : 1_by_n_numpy_array,
                                    'mse_std'        : 1_by_n_numpy_array,
                                    'rmse_rads_mean': 1_by_n_numpy_array,
                                    'rmse_rads_std' : 1_by_n_numpy_array,
                                    ...
                               },
                    'val'  : {...},
                    'test' : {...}
                }

               For example, you can access the MSE results for the␣
↪validation
               set like so:
                   results['train'][metric]
               For example, you can access the summary (i.e. the average␣
↪results
               over all the rotations) for the test set for the rMSE in␣
↪degrees
               like so:
                   summary['test']['rmse_degs_mean']
        '''

        # Verify a valid train set size was provided
        nfolds = len(all_Xfolds)
        if trainsize > nfolds - 2:
```

```
            err_msg = "ERROR: KFoldHolisticCrossValidation.
↪perform_cross_validation() - "
            err_msg += "trainsize (%d) cant be more than nfolds (%d) - 2" %␣
↪(trainsize, nfolds)
            raise ValueError(err_msg)

        # Set up results recording for each rotation
        results = {'train': None, 'val': None, 'test': None}
        summary = {'train': {}, 'val': {}, 'test': {}}

        model = self.model
        evaluate = self.eval_func

        # TODO: Rotate through the data to try different train, val, and test␣
↪sets
        for rotation in range(0, nfolds, self.rotation_skip):
            # TODO: Determine fold indices for train, val, and test set.
            #       The val and tests are each only 1 fold
            trainfolds = (np.arange(0, trainsize)+rotation)%nfolds # TODO
            valfold = (nfolds-2+rotation)%nfolds# TODO
            testfold = (nfolds-1+rotation)%nfolds# TODO

            # TODO: Construct train set by concatenating the individual␣
↪training
            #       folds together (hint: see np.take() and np.concatenate())
            X = np.concatenate([all_Xfolds[trainfold] for trainfold in␣
↪trainfolds])# TODO
            y = np.concatenate([all_yfolds[trainfold] for trainfold in␣
↪trainfolds])# TODO

            # TODO: Construct validation set. Hint: this is always one fold
            Xval = all_Xfolds[valfold]# TODO
            yval = all_yfolds[valfold]# TODO

            # TODO: Construct test set
            Xtest = all_Xfolds[testfold]# TODO
            ytest = all_yfolds[testfold]# TODO

            # DEBUGGING
            if verbose:
                print("TRAIN", X.shape, y.shape, trainfolds)
                print("VAL", Xval.shape, yval.shape, valfold)
                print("TEST", Xtest.shape, ytest.shape, testfold)

            # TODO: Train model using the training set
            model.fit(X,y)
```

```
            # TODO: Predict with the model for train, val, and test sets
            preds = model.predict(X)#TODO
            preds_val = model.predict(Xval)#TODO
            preds_test = model.predict(Xtest)#TODO

            # TODO: Evaluate the model for each set
            res_train = evaluate(model, X,y,preds)#TODO
            res_val = evaluate(model, Xval,yval,preds_val)#TODO
            res_test = evaluate(model, Xtest,ytest,preds_test)#TODO
#             print(res_train)

            # Record the train, val, and test set results. These are dicts
            # of result metrics, returned by the evaluate function
            # TODO: For the first rotation, store the results from evaluating
            #       with the train, val, and tests by setting the values of the␣
↪
            #       appropriate items within the results dict
            if results['train'] is None:
                results['train'] = res_train#TODO
                results['val'] = res_val#TODO
                results['test'] = res_test#TODO
            else:
                # Append the results for each rotation
                for metric in res_train.keys():
                    results['train'][metric] = np.
↪append(results['train'][metric],
                                                    res_train[metric],␣
↪axis=0)
                    results['val'][metric] = np.append(results['val'][metric],
                                                  res_val[metric], axis=0)
                    results['test'][metric] = np.append(results['test'][metric],
                                                   res_test[metric],␣
↪axis=0)
#             print(results)

        # Compute and record the mean and standard deviation for the given size␣
↪for each metric
        for metric in results['train'].keys():
            for stat_set in ['train', 'val', 'test']:
                summary[stat_set][metric+'_mean'] = np.
↪mean(results[stat_set][metric],
                                                      axis=0).reshape(1,␣
↪-1)
                summary[stat_set][metric+'_std'] = np.
↪std(results[stat_set][metric],
```

```
                                                      axis=0).reshape(1, -1)

        return results, summary

    def grid_cross_validation(self, all_Xfolds, all_yfolds, verbose=0):
        ''' TODO
        (MAIN PROCEDURE) Perform cross validation for multiple sets of
        parameters and train set sizes. Calls self.perform_cross_validation().
        This is the procedure that executes cross validation for all parameter
        sets and all sizes.

        PARAMS:
            all_Xfolds: all the input data folds (list of folds, as it was
                        loaded from the files)
            all_yfolds: all the output data folds (list of folds)
            verbose: flag to print out simple debugging information

        RETURNS: best parameter set for each train set size as a list of
                 parameter indices. Additionally, returns self.report_by_size,
                 the 3D array of validation means (overall rotations) for all
                 paramsets, for each metric, for all sizes. The structure of
                 the returned object is a dictionary of the following form:
                 {
                   'report_by_size' : self.report_by_size,
                   'best_param_inds': self.best_param_inds
                 }
        '''
        sizes = self.trainsizes
        paramsets = self.paramsets
        nparamsets = len(paramsets)
        print("nparamsets", nparamsets)

        # Set up all results
        all_results = []

        # Iterate over parameter sets
        for params in paramsets:
            # Set up paramset results
            param_res = []
            param_smry = None

            # Set model parameters
            print("Current paramset\n", params)
            self.model.set_params(**params)

            # Iterate over the different train set sizes
            for size in sizes:
```

```
                # TODO: Cross-validation for the current model and train set␣
↪size
                res, smry = self.
↪perform_cross_validation(all_Xfolds,all_yfolds, size,verbose)# TODO

                # Save the results
                param_res.append(res)
                # Save the mean and standard deviation statistics (summary)
                if param_smry is None: param_smry = smry
                else:
                    # For each metric measured, append the summary results
                    for metric in smry['train'].keys():
                        for stat_set in ['train', 'val', 'test']:
                            stat = smry[stat_set][metric]
                            param_smry[stat_set][metric] = np.
↪append(param_smry[stat_set][metric],
                                                                        stat,␣
↪axis=0)

            # Append the results and summary for the parameter set
            all_results.append({'params':params, 'results':param_res,
                                'summary':param_smry})

        # Generate reports and determine best params for each size
        self.results = all_results
        self.report_by_size = self.get_reports()
        self.best_param_inds = self.get_best_params(self.opt_metric,
                                                    self.maximize_opt_metric)
        return {'report_by_size':self.report_by_size,
                'best_param_inds':self.best_param_inds}

    def get_reports(self):
        ''' PROVIDED
        Get the mean validation summary of all the parameters for each size
        for all metrics. This is used to determine the best parameter set
        for each size

        RETURNS: the report_by_size as a 3D s-by-r-by-p array. Where s is
                 the number of train sizes tried, r is the number of summary
                 metrics evaluated+2, and p is the number of parameter sets.
        '''
        results = self.results
        sizes = np.reshape(self.trainsizes, (1, -1))

        nsizes = sizes.shape[1]
        nparams = len(results)
```

```
        # Set up the reports objects
        metrics = list(results[0]['summary']['val'].keys())
        colnames = ['params', 'size'] + metrics
        report_by_size = np.empty((nsizes, len(colnames), nparams),␣
↪dtype=object)

        # Determine the mean val for each parameter set for each size for all␣
↪metrics
        for p, paramset_result in enumerate(results):
            params = paramset_result['params']
            res_val = paramset_result['summary']['val']

            # Compute the mean val performance for each train size for each␣
↪metric
            means_by_size = [np.mean(res_val[metric], axis=1) for metric in␣
↪metrics]
            # Include the train set sizes into the report
            means_by_size = np.append(sizes, means_by_size, axis=0)
            # Include the parameter sets into the report
            param_strgs = np.reshape([str(params)]*nsizes, (1, -1))
            means_by_size = np.append(param_strgs, means_by_size, axis=0).T
            # Append the parameter set means into the report
            report_by_size[:,:,p] = means_by_size
        return report_by_size

    def get_best_params(self, opt_metric, maximize_opt_metric):
        ''' PROVIDED (Do read through all the provided code)
        Determines the best parameter set for each train size, based
        on a specific metric.

        PARAMS:
            opt_metric: optimized metric. one of the metrics returned
                        from eval_func, with '_mean' appended for the
                        summary stat. This is the mean metric used to
                        determine the best parameter set for each size

            maximize_opt_metric: True if the max of opt_metric should be
                                 used to determine the best parameters.
                                 False if the min should be used.
        RETURNS: list of best parameter set indicies for each size
        '''
        results = self.results
        report_by_size = self.report_by_size

        metrics = list(results[0]['summary']['val'].keys())

        # Determine best params for each size, for the optimized metric
```

```
        best_param_inds = None
        metric_idx = metrics.index(opt_metric)

        if maximize_opt_metric:
            # Add two for the additional cols for params and size
            best_param_inds = np.argmax(report_by_size[:, metric_idx+2, :],␣
↪axis=1)
        else:
            best_param_inds = np.argmin(report_by_size[:, metric_idx+2, :],␣
↪axis=1)
        # Return list of best params indices for each size
        return best_param_inds

    def get_best_params_strings(self):
        ''' PROVIDED
        Generates a list of strings of the best params for each size
        RETURNS: list of strings of the best params for each size
        '''
        best_param_inds = self.best_param_inds
        results = self.results
        return [str(results[p]['params']) for p in best_param_inds]

    def get_report_best_params_for_size(self, size):
        ''' PROVIDED
        Get the mean validation summary for the best parameter set
        for a specific size for all metrics.
        PARAMS:
            size: index of desired train set size for the best
                  paramset to come from. Size here is the index in
                  the trainsizes list, NOT the actual number of folds.
        RETURNS: the best parameter report for the size as an s-by-m
                 dataframe. Where each row is for a different size, and
                 each column is for a different summary metric.
        '''
        best_param_inds = self.best_param_inds
        report_by_size = self.report_by_size

        bp_index = best_param_inds[size]

        metrics = list(self.results[0]['summary']['val'].keys())
        colnames = ['params', 'size'] + metrics
        report_best_params_for_size = pd.DataFrame(report_by_size[:,:,bp_index],
                                                   columns=colnames)
        return report_best_params_for_size

    def plot_cv(self, foldsindices, results, summary, metrics, size):
        ''' PROVIDED
```

```
    Plotting function for after perform_cross_validation(),
    displaying the train and val set performances for each rotation
    of the training set.

    PARAMS:
        foldsindices: indices of the train sets tried
        results: results from perform_cross_validation()
        summary: mean and standard deviations of the results
        metrics: list of result metrics to plot. Available metrics
                 are the keys in the dict returned by eval_func
        size: train set size

    RETURNS: the figure and axes handles
    '''
    nmetrics = len(metrics)

    # Initialize figure plots
    fig, axs = plt.subplots(nmetrics, 1, figsize=(12,6))
    fig.subplots_adjust(hspace=.4)
    # When 1 metric is provided, allow the axs to be iterable
    axs = np.array(axs).ravel()

    # Construct each subplot
    for metric, ax in zip(metrics, axs):
        # Compute the mean for multiple outputs
        res_train = np.mean(results['train'][metric], axis=1)
        res_val = np.mean(results['val'][metric], axis=1)
        # Plot
        ax.plot(foldsindices, res_train, label='train')
        ax.plot(foldsindices, res_val, label='val')
        ax.set(ylabel=metric)
    axs[0].legend(loc='upper right')
    axs[0].set(xlabel='Fold Index')
    axs[0].set(title='Performance for Train Set Size ' + str(size))
    return fig, axs

def plot_param_train_val(self, metrics, paramidx=0, view_test=False):
    ''' PROVIDED
    Plotting function for after grid_cross_validation(),
    displaying the mean (summary) train and val set performances
    for each train set size.

    PARAMS:
        metrics: list of summary metrics to plot. '_mean' or '_std'
                 must be append to the end of the base metric name.
                 These base metric names are the keys in the dict
                 returned by eval_func
```

```
            paramidx: parameter set index
            view_test: flag to view the test set results

        RETURNS: the figure and axes handles
        '''
        sizes = self.trainsizes
        results = self.results

        summary = results[paramidx]['summary']
        params = results[paramidx]['params']

        nmetrics = len(metrics)

        # Initialize figure plots
        fig, axs = plt.subplots(nmetrics, 1, figsize=(12,6))
        fig.subplots_adjust(hspace=.4)
        # When 1 metric is provided, allow the axs to be iterable
        axs = np.array(axs).ravel()

        # Construct each subplot
        for metric, ax in zip(metrics, axs):
            # Compute the mean for multiple outputs
            res_train = np.mean(summary['train'][metric], axis=1)
            res_val = np.mean(summary['val'][metric], axis=1)
            # Plot
            ax.plot(sizes, res_train, label='train')
            ax.plot(sizes, res_val, label='val')
            if view_test:
                res_test = np.mean(summary['test'][metric], axis=1)
                ax.plot(sizes, res_test, label='test')
            ax.set(ylabel=metric)
        axs[-1].set(xlabel='Train Set Size (# of folds)')
        axs[0].set(title=str(params))
        axs[0].legend(loc='upper right')
        return fig, axs

    def plot_allparams_val(self, metrics):
        ''' PROVIDED
        Plotting function for after grid_cross_validation(), displaying
        mean (summary) validation set performances for each train size
        for all parameter sets for the specified metrics.

        PARAMS:
            metrics: list of summary metrics to plot. '_mean' or '_std'
                     must be append to the end of the base metric name.
                     These base metric names are the keys in the dict
                     returned by eval_func
```

```
        RETURNS: the figure and axes handles
        '''
        sizes = self.trainsizes
        results = self.results

        nmetrics = len(metrics)

        # Initialize figure plots
        fig, axs = plt.subplots(nmetrics, 1, figsize=(10,6))
        fig.subplots_adjust(hspace=.4)
        # When 1 metric is provided, allow the axs to be iterable
        axs = np.array(axs).ravel()

        # Construct each subplot
        for metric, ax in zip(metrics, axs):
            for p, param_results in enumerate(results):
                summary = param_results['summary']
                params = param_results['params']
                # Compute the mean for multiple outputs
                res_val = np.mean(summary['val'][metric], axis=1)
                ax.plot(sizes, res_val, label=str(params))
            ax.set(ylabel=metric)
        axs[-1].set(xlabel='Train Set Size (# of folds)')
        axs[0].set(title='Validation Performance')
        axs[0].legend(bbox_to_anchor=(1.02, 1), loc='upper left',
                      ncol=1, borderaxespad=0., prop={'size': 8})
        return fig, axs

    def plot_best_params_by_size(self):
        ''' PROVIDED
        Plotting function for after grid_cross_validation(), displaying
        mean (summary) train and validation set performances for the best
        parameter set for each train size for the specified metrics.

        RETURNS: the figure and axes handles
        '''
        results = self.results
        metric = self.opt_metric
        best_param_inds = self.best_param_inds
        sizes = np.array(self.trainsizes)

        # Unique set of best params for the legend
        unique_param_sets = np.unique(best_param_inds)
        lgnd_params = [self.paramsets[p] for p in unique_param_sets]

        # Initialize figure
```

```
    fig, axs = plt.subplots(2, 1, figsize=(10,6))
    fig.subplots_adjust(hspace=.4)
    # When 1 metric is provided, allow the axs to be iterable
    axs = np.array(axs).ravel()
    set_names = ['train', 'val']

    # Construct each subplot
    for i, (ax, set_name) in enumerate(zip(axs, set_names)):
        for p in unique_param_sets:
            # Obtain indices of sizes this paramset was best for
            param_size_inds = np.where(best_param_inds == p)[0]
            param_sizes = sizes[param_size_inds]
            # Compute the mean over multiple outputs for each size
            param_summary = results[p]['summary'][set_name]
            metric_scores = np.mean(param_summary[metric][param_size_inds, :
↪], axis=1)
            # Plot the param results for each size it was the best for
            ax.scatter(param_sizes, metric_scores, s=120, marker=(p+2, 1))
            #ax.grid(True)

        set_name += ' Set Performance'
        ax.set(ylabel=metric, title=set_name)

    axs[-1].set(xlabel='Train Set Size (# of folds)')
    axs[0].legend(lgnd_params, bbox_to_anchor=(1.02, 1), loc='upper left',
                  ncol=1, borderaxespad=0., prop={'size': 8})
    return fig, axs
```

6 PERFORM CROSS VALIDATION FOR ELASTICNET

[9]:
```
""" TODO
Generate list of parameters to use for cross validation
using generate_paramsets()
"""
param_lists = {'alpha':[.001, .005, .01, .05, .1],
               'l1_ratio':[.05, .1], 'max_iter':[1e4]}

allparamsets = generate_paramsets(param_lists)# TODO
allparamsets
```

[9]:
```
[{'alpha': 0.001, 'l1_ratio': 0.05, 'max_iter': 10000.0},
 {'alpha': 0.001, 'l1_ratio': 0.1, 'max_iter': 10000.0},
 {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0},
 {'alpha': 0.005, 'l1_ratio': 0.1, 'max_iter': 10000.0},
 {'alpha': 0.01, 'l1_ratio': 0.05, 'max_iter': 10000.0},
```

```
 {'alpha': 0.01, 'l1_ratio': 0.1, 'max_iter': 10000.0},
 {'alpha': 0.05, 'l1_ratio': 0.05, 'max_iter': 10000.0},
 {'alpha': 0.05, 'l1_ratio': 0.1, 'max_iter': 10000.0},
 {'alpha': 0.1, 'l1_ratio': 0.05, 'max_iter': 10000.0},
 {'alpha': 0.1, 'l1_ratio': 0.1, 'max_iter': 10000.0}]
```

[10]:
```
""" TODO
Initialize the cross validation object. Use ElasticNet for the
ase model, use every even value between 2 and 18, inclusive, for
the train set sizes, use score_eval as the eval_func, use rmse_degs
as the metric to optimize, and 4 for the skip. We want ot minimize
rmse thus set maximize_opt_metrix=False
"""
model = ElasticNet()# TODO
trainsizes = range(2,20,2)# TODO
opt_metric = 'rmse_degs'# TODO
maximize_opt_metric = False# TODO
skip = 4# TODO
crossval = KFoldHolisticCrossValidation(model, allparamsets, score_eval,␣
↪opt_metric,
                                        maximize_opt_metric, trainsizes, skip)
```

[11]:
```
""" TODO
Execute the grid_cross_validation() procedure for all parameters
and train set sizes
"""
# TODO: make sure this is set appropriately. True if you want to
#       just always to run cross validation, false if you want
#       to re-load a previous run
force = False
fullcvfname = "hw6_crossval.pkl"

crossval_report = None
if force or (not os.path.exists(fullcvfname)):
    # TODO: Use grid_cross_validation() to run the full cross
    #       validation procedure
    # Note: when testing, run this using small lists of parameters
    #       (e.g. of length 2 or 4) and/or small trainsize lists
    #       (e.g. [1, 2, 3, 4, 5])
    # Note: for the final submission, make sure to use the complete
    #       parameter set list and trainsize list provided/specified
    #       This will take some time.
    crossval_report = crossval.grid_cross_validation(MI_folds, torque_folds,␣
↪verbose=1)#TODO
    joblib.dump(crossval, fullcvfname)
else:
    # TODO: Re-load saved crossval object instead of re-running the
```

```python
    #       cross validation procedure. Use joblib.load()
    crossval = joblib.load(fullcvfname)# TODO
    crossval_report = {'report_by_size' : crossval.report_by_size,
                       'best_param_inds': crossval.best_param_inds}

crossval_report.keys()
```

[11]: dict_keys(['report_by_size', 'best_param_inds'])

7 RESULTS

[12]:
```python
""" TODO
Obtain all the results for all parameters, for all sizes, for all
rotations. This is the results attribute of the crossval object
"""
all_results = crossval.results # TODO
len(all_results)
```

[12]: 10

[13]:
```python
""" PROVIDED
Display the keys of the results object
"""
all_results[0].keys()
```

[13]: dict_keys(['params', 'results', 'summary'])

[14]:
```python
""" TODO
Obtain and display the indices of the best parameters for each
size using either the best_params_inds attribute of the crossval
object or 'best_param_inds' item from the crossval_report dict
"""
best_param_inds = crossval.best_param_inds# TODO
best_param_inds
```

[14]: array([5, 4, 4, 2, 2, 2, 2, 2, 2])

[15]:
```python
""" TODO
Display the list of the best parameter sets for each size. Use
crossval.get_best_params_strings()
"""
# TODO
crossval.get_best_params_strings()
```

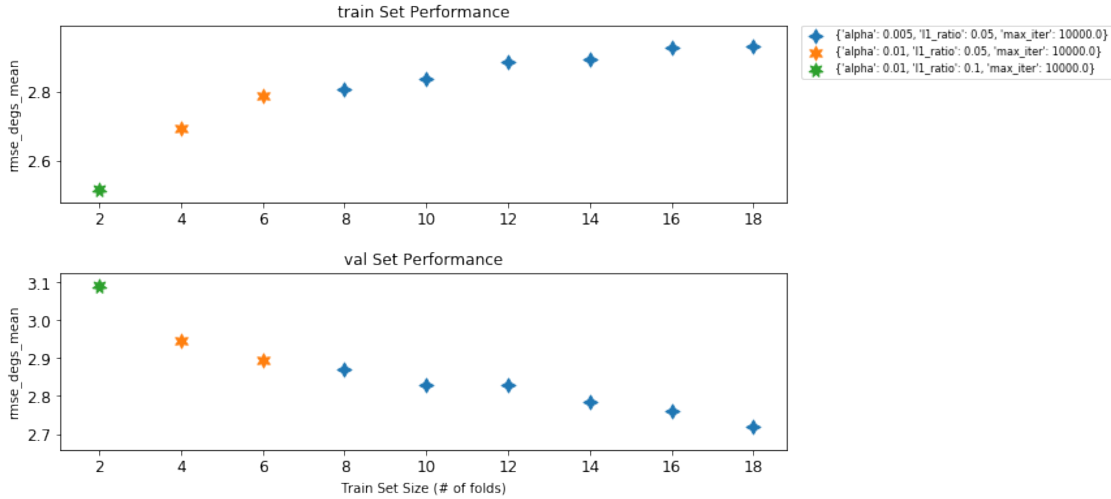
[15]:
```
["{'alpha': 0.01, 'l1_ratio': 0.1, 'max_iter': 10000.0}",
 "{'alpha': 0.01, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.01, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0}",
 "{'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter': 10000.0}"]
```

[16]:
```
""" TODO
Obtain and dsplay the shape of the report of all the parameters'
mean results over all sizes and rotations. This is the report_by_size
attribute of the crossval object. It is also stored within the
'report_by_size' item of the crossval_report dict
"""
report = crossval_report['report_by_size']# TODO
report.shape
```

[16]:
```
(9, 12, 10)
```

[17]:
```
""" 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()
"""
# TODO
crossval.plot_best_params_by_size()
```

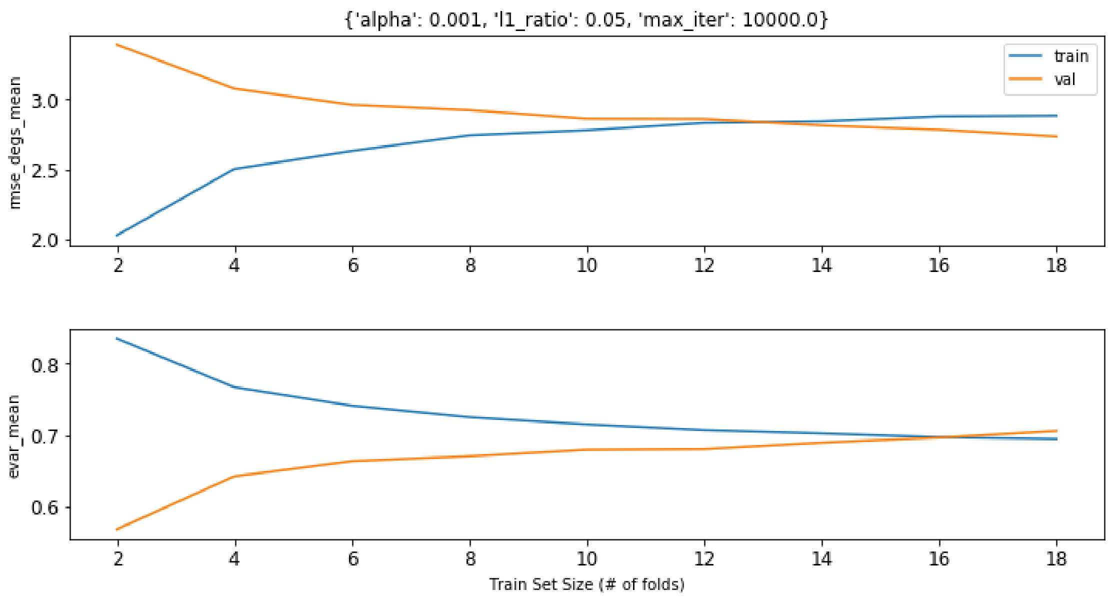
[17]:
```
(<Figure size 720x432 with 2 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c73acc18>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c73f6b38>],
       dtype=object))
```



[18]:
```
""" TODO
Plot the average results (summary) over train set size for all
parameter sets for the metrics 'rmse_degs_mean' and 'evar_mean'
for the train and val sets. Use plot_param_train_val().
view_test=False
"""
metrics = ['rmse_degs_mean', 'evar_mean']

# TODO
crossval.plot_param_train_val(metrics)
```

[18]:
```
(<Figure size 864x432 with 2 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c7494fd0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c71ae5f8>],
       dtype=object))
```

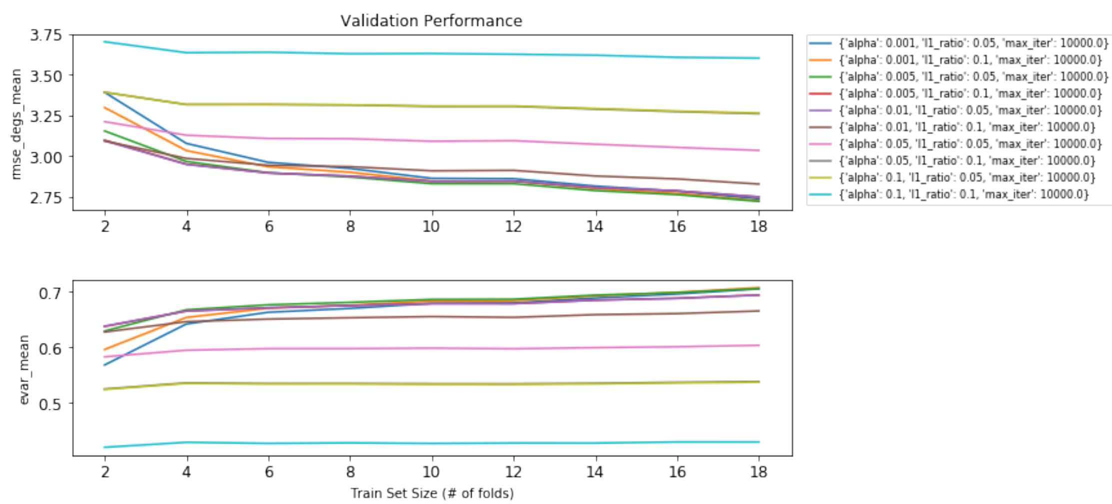


[19]:

```python
""" TODO
Plot the validation results for all parameters over all train
sizes, for the specified metrics. Use plot_allparams_val()
"""
# TODO
crossval.plot_allparams_val(metrics)
```

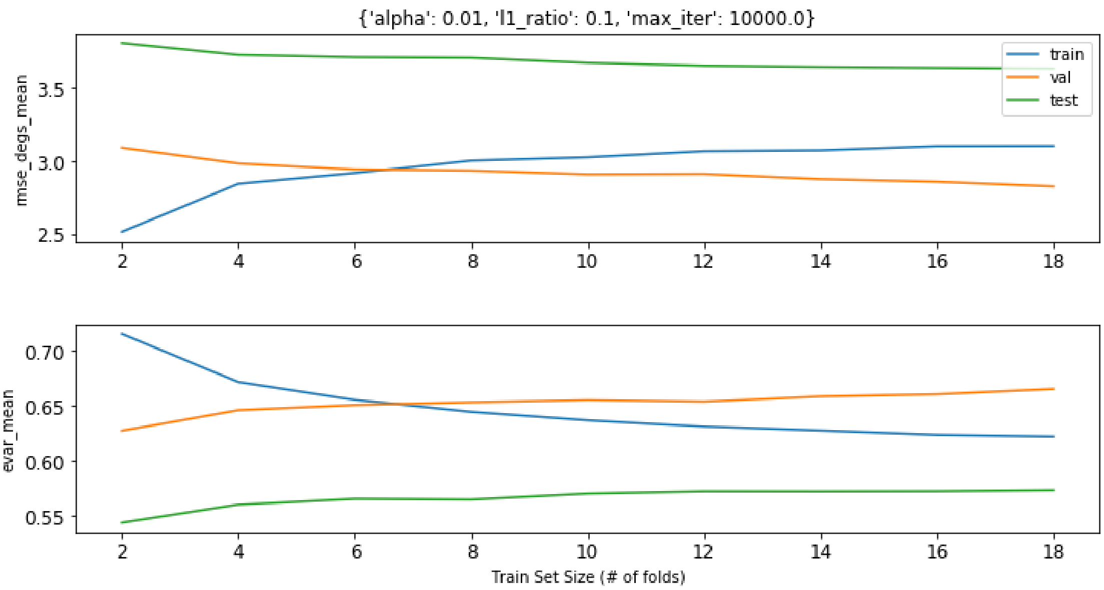
[19]:

```
(<Figure size 720x432 with 2 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c7131ba8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c70f12b0>],
       dtype=object))
```



[22]:
```
""" TODO
For the best parameter set for the train set size at index 5,
plot the TRAIN, VAL, and TEST set performances using
plot_param_train_val() for just the optimized metric
"""
size_idx = 5
# TODO
crossval.plot_param_train_val(metrics, paramidx= size_idx, view_test= True)
```

[22]:
```
(<Figure size 864x432 with 2 Axes>,
 array([<matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c6d9b9b0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7ff1c6d667b8>],
       dtype=object))
```



[21]:
```
""" TODO
Use get_report_best_params_for_size() to display the report of
the average val statistics for the best parameter set, for the
train set size at index 5 (i.e. size_idx)
"""
report_best_params = crossval.get_report_best_params_for_size(size_idx)# TODO
report_best_params
```

[21]:
```
                                              params  size  \
0  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…   2.0
1  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…   4.0
2  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…   6.0
```

```
3  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…   8.0
4  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…  10.0
5  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…  12.0
6  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…  14.0
7  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…  16.0
8  {'alpha': 0.005, 'l1_ratio': 0.05, 'max_iter':…  18.0

               mse_mean                mse_std        rmse_rads_mean  \
0  0.003821254929479434  0.0015570898371603686  0.055036647454143876
1  0.003359794724669028  0.0013176979170185575  0.051756061138804495
2  0.003161254162651294  0.0011999152541327309    0.0505443371326435
3   0.00308032158618960   0.0010843591479785413   0.05008068785619305
4   0.00298101427747241   0.0010667836268901884   0.04936768908176648
5   0.00297921172306356   0.00107059331231420     0.04936472238622769
6  0.002891813019455461   0.00109625486295136     0.04862113979301079
7  0.002836969267130076   0.00106868619949819   0.048185134965864386
8  0.002743245615294886   0.00100114960466126     0.04746171148551416

          rmse_rads_std      rmse_degs_mean       rmse_degs_std  \
0  0.011650748560957552   3.153367617671871  0.6675387207109851
1  0.010767569940499959  2.9654038674745506  0.6169363132025786
2  0.009918001086182373  2.8959771959868412  0.5682596034444163
3  0.009159891154794488  2.8694120492719364   0.524823103968938
4   0.00909524365761089   2.828560228699294   0.521119075224234
5  0.009060773899829252   2.828390249565821   0.519144103582508
6  0.009254594858360747   2.785786105255099  0.5302492264875427
7  0.009061403439161595  2.7608048688122793  0.5191801735292887
8  0.008728666016444871   2.719355756587546  0.5001157235215599

             evar_mean             evar_std          score_mean  \
0  0.6294533959163975  0.06037145147259193  0.7672804166442525
1  0.6678243717788315  0.05605088317763063  0.7940617853408632
2  0.6768776099665942  0.03625709358086175  0.8062494279037515
3  0.6812768063724152  0.03437172476571209  0.8100478693457115
4  0.6866205223251372  0.03679730423636094  0.8163772513886203
5  0.6869903461770769  0.03904989367498715   0.816309018609658
6  0.6943689167990303  0.03899678551075173  0.8221938773769729
7  0.6989297636361852  0.03781438154300826  0.8253320616810855
8    0.70617812438989  0.03556165997292166  0.8308623883411486

              score_std
0  0.055363943421294054
1    0.0511287908565875
2   0.03643982248736823
3   0.02928359638640827
4  0.028263055862259936
5   0.02753062051815879
```

```
6  0.029631746845968115
7  0.029077945120516882
8  0.026332667557775403
```