Assignment 3 Advanced Programming

Authors: Rafaela Becerra & Marta Cortés ¶

Part I Scikit-Learn

Preprocess

Some libraries are imported and the solar dataset is read (in cvs format) into a Pandas dataframe

```
In [1]: import numpy as np
        from numpy.random import randint
        import pandas as pd

        import os

        os.chdir('/Users/cortesocanamarta/Documents/Marta/MÁSTER DATA SCIEN
        CE/Advanced Programming/Third Assignment')
        os.getcwd()

        train = pd.read_csv('train.csv')
        test = pd.read_csv('test.csv')
```

NIA as randomseed in order to create reproductible results:

```
In [2]: my_NIA = 1722552880
        np.random.seed(my_NIA)
```

Determine the response variable and atributes for the train and test set

```
In [3]: X_train=train[train.columns[0:1200]]
        X_test=test[test.columns[0:1200]]

        y_train=train['energy']
        y_test=test['energy']
```

Normalization of the atributes with MinMaxScaler, which allows to transform these features by scaling each to a range between 0 and 1.

```
In [4]: from sklearn import preprocessing
        min_max_scaler = preprocessing.MinMaxScaler()
        X_train_minmax = min_max_scaler.fit_transform(X_train)
        X_test_minmax = min_max_scaler.transform(X_test)
```

Selection of the first 75 columns which represent one blue point features

```
In [5]: X_train_minmax=X_train_minmax[:,0:75]
        X_test_minmax=X_test_minmax[:,0:75]
```

Create a validation partition, the first 10 years data for trainning and the rest for the validation process

```
In [6]: from sklearn.model_selection import PredefinedSplit
        validation_indices = np.zeros(X_train_minmax.shape[0])
        validation_indices[:round(10/12*X_train_minmax.shape[0])] = -1
        tr_val_partition = PredefinedSplit(validation_indices)
```

Machine Learning methods with default parameters

We have selected the mean squared error as a metric for comparing the different kind of machine learning methods. The mean squared error or MSE, simultaneously minimizes the prediction variance and prediction bias, then, it is a good quality measure of an estimator. The lower the MSE, it will mean a closer estimation of the quantity of energy produce compare to the real data. The MSE can be expressed as:

$$MSE = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Regression Models

Decision Trees are a non-parametric supervised learning method that can be used for classification and regression. In this case, the goal is to create a model that predicts the value of the target variable which is the amount of energy produce, by learning simple decision rules inferred from the data atributes.

```
In [7]: from sklearn import tree
        rgt = tree.DecisionTreeRegressor()
        rgt.fit(X_train_minmax, y_train)

        y_test_pred = rgt.predict(X_test_minmax)

        from sklearn import metrics
        from sklearn.metrics import mean_squared_error

        print("\n The MSE is:\n", metrics.mean_squared_error(y_test_pred, y
        _test))
```

```
 The MSE is:
 21030163731252.54
```

KNN

The Neighbors-based regression is used when the data target is continuous rather tha discrete variable, then the variable is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

```
In [8]: from sklearn.neighbors import KNeighborsRegressor
        neigh = KNeighborsRegressor()
        neigh.fit(X_train_minmax, y_train)

        y_test_pred_KNN = neigh.predict(X_test_minmax)

        print("\n The MSE is:\n", metrics.mean_squared_error(y_test_pred_KN
        N, y_test))
```

```
 The MSE is:
 12439134529957.766
```

SVMs

Support vector machines are a set of supervised learning methods used for classification, regression and outlier detection. In the case of regression, this method approximates the best values with a given margin called epsilon, that considers the model complexity and error rate.

```
In [9]: from sklearn import svm
        svmr = svm.SVR(gamma='auto')
        svmr.fit(X_train_minmax, y_train)

        y_test_pred_SVM = svmr.predict(X_test_minmax)

        print("\n The MSE is:\n", metrics.mean_squared_error(y_test_pred_SV
        M, y_test))
```

```
 The MSE is:
 56795190527414.45
```

Machine Learning methods with tuning hyper-parameters

Hyperparameter optimization or tuning is the problem of choosing a set of optimal hyperparameters for a learning algorithm. Each method has important parameters that can be change in order to get the best model. It is necesary to define:

- The search space (the hyper-parameters of the method and their allowed values): these will differ for each method.
- The search method: in this case for all the methods will be a random-search.
- The evaluation method: it will be use a crossvalidation process.

Regression Trees

In regression trees, the parameters that will be optimized are:

- max_depth (default=None):the maximum depth of the tree.
- min_samples_split (default=2): the minimum number of samples required to split an internal node

Since, both are discrete variables sp_randint is used in order to create a discrete uniform distribution. Hence, the default parameters are:

- criterion (default='mse'). The function to measure the quality of a split.
- splitter (default='best'). The strategy used to choose the split at each node.
- min_samples_leaf (default=1). The minimum number of samples required to be at a leaf node.
- min_weight_fraction_leaf (default=0.). The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node.
- random_state (default=None). The random number generator is the RandomState instance used by np.random.
- max_leaf_nodes (default=None). Unlimited number of nodes.
- min_impurity_decrease (default=0.). A node will be split if this split induces a decrease of the impurity greater than or equal to 0.
- min_impurity_split (default=1e-7). Threshold for early stopping in tree growth.

In [10]:
```
from scipy.stats import randint as sp_randint

param_grid={'max_depth': sp_randint(2,20), 'min_samples_split':sp_r
andint(2,20)}

from sklearn.model_selection import RandomizedSearchCV
rgt_grid = RandomizedSearchCV(rgt, param_grid, scoring='neg_mean_sq
uared_error', cv=tr_val_partition, n_jobs=1, verbose=1)

rgt_grid.fit(X_train_minmax, y_train)

y_test_pred_rgt_G = rgt_grid.predict(X_test_minmax)

print("\n The best parameters across all the searched parameters:\n
", rgt_grid.best_params_)
print("\n The MSE is:\n", metrics.mean_squared_error(y_test_pred_rg
t_G, y_test))
```

```
Fitting 1 folds for each of 10 candidates, totalling 10 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.

 The best parameters across all the searched parameters:
 {'max_depth': 4, 'min_samples_split': 12}

 The MSE is:
 13452821463108.81

[Parallel(n_jobs=1)]: Done  10 out of  10 | elapsed:    1.8s finis
hed
```

KNN

The hyper-parameter that needs to be optimized at least is the 'n_neighbors'(default = 5), which corresponds to the number of neighbors that will be used set to a maximun of 20. The rest of the parameters will be set as default:

- weights= 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- p= 2 : use of euclidean_distance for calculation.
- algorithm: 'auto'. Decides the most appropriate algorithm based on the values passed to fit method.
- leaf_size= 30 : leaf size passed when BallTree or KDTree algorith are use.
- metric= 'minkowski'. The distance metric to use for the tree.
- metric_params= None. Zero additional keyword arguments for the metric function.

```
In [11]: param_grid={'n_neighbors': sp_randint(1,20)}

         neigh_grid = RandomizedSearchCV(neigh, param_grid, scoring='neg_mea
         n_squared_error', cv=tr_val_partition, n_jobs=1, verbose=1)

         neigh_grid.fit(X_train_minmax, y_train)

         y_test_pred_KNN_G = neigh_grid.predict(X_test_minmax)

         print("\n The best parameters across all the searched parameters:\n
         ", neigh_grid.best_params_)
         print("\n The MSE is:\n", metrics.mean_squared_error(y_test_pred_KN
         N_G, y_test))
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.

Fitting 1 folds for each of 10 candidates, totalling 10 fits

[Parallel(n_jobs=1)]: Done  10 out of  10 | elapsed:    1.6s finis
hed

 The best parameters across all the searched parameters:
 {'n_neighbors': 19}

 The MSE is:
 10686112346565.672
```

SVMs

The hyper-parameters choosen to be optimized are:

- kernel (default='rbf'): specifies the kernel type to be used in the algorithm. Three types of possible kernels will be consider. These are 'linear', 'poly', and 'rbf'.
- degree (default=3): degree of the polynomial kernel function if 'poly' is consider. This will be set as a integer sequence of possible values, with a maximun of 6.
- C (default=1.0): regularization parameter. It controls the trade off between smooth decision boundary and classifying the training points correctly. C in svm, is not too sensitive, then it is quicker to try different values that come from a logarithmic scale.
- gamma (default='scale'): Kernel coefficient for 'rbf', and 'poly', it will be optimized as a float, considering a sequence set as a maximum of 0.1.
- epsilon (default=0.1): Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value. It will be optimized as a float, considering various options with a maximun set as 0.9.

In contrast, the parameters set as default for this method are:

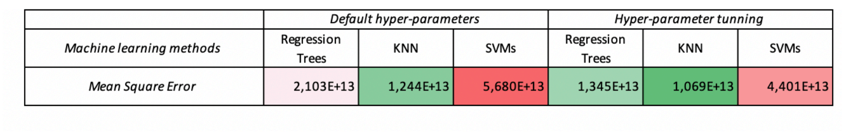
- shrinking (default=True).
- max_iter (default=-1). No limit on iterations within solver.

```
In [12]: param_grid1={'C': np.logspace(0, 3, num=10000), 'gamma': [1e-7, 1e-
         1], 'kernel': ('linear', 'rbf','poly'), 'degree' : [0, 1, 2, 3, 4,
         5, 6], 'epsilon':[0.1,0.9]}

         svmr_grid = RandomizedSearchCV(svmr, param_grid1, scoring='neg_mean
         _squared_error', cv=tr_val_partition, n_jobs=1, verbose=1)

         svmr_grid.fit(X_train_minmax, y_train)

         y_test_pred_SVM_G = svmr_grid.predict(X_test_minmax)

         print("\n The best parameters across all the searched parameters:\n
         ", svmr_grid.best_params_)
         print("\n The MSE is:\n", metrics.mean_squared_error(y_test_pred_SV
         M_G, y_test))
```

```
Fitting 1 folds for each of 10 candidates, totalling 10 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.
[Parallel(n_jobs=1)]: Done  10 out of  10 | elapsed:   13.9s finis
hed

 The best parameters across all the searched parameters:
 {'kernel': 'linear', 'gamma': 1e-07, 'epsilon': 0.1, 'degree': 3,
'C': 331.09452188119764}

 The MSE is:
 44014839134973.82
```

```
In [ ]:
```

From all the results that we got, we can say first that all the models when optimizing its hyper-paremeters, the metrics result better, and give a more accurate model. By comparing the MSE resulting between type of methods, the one that is more accurate is the KNN method, which returned a better metric than the regression tree method and the support vector regressor (Table 1). These results will be much more accurate, if all the data is use and not only the variables from one of the points.



	Default hyper-parameters			*Hyper-parameter tunning*		
Machine learning methods	Regression Trees	KNN	SVMs	Regression Trees	KNN	SVMs
Mean Square Error	2,103E+13	1,244E+13	5,680E+13	1,345E+13	1,069E+13	4,401E+13

Part II Scikit-Learn

Preprocess

Selection of the first 300 columns which represent four blue point features

```
In [ ]:
```

```
In [13]: X_train=train[train.columns[0:300]]
         X_test=test[test.columns[0:300]]
```

Create a validation partition, the first 10 years data for trainning and the rest for the validation process

```
In [14]: from sklearn.model_selection import PredefinedSplit
         validation_indices = np.zeros(X_train.shape[0])
         validation_indices[:round(10/12*X_train.shape[0])] = -1
         tr_val_partition = PredefinedSplit(validation_indices)
```

Selection of the 10% of attributes and random assignation of 10% of nan for each column in the train and test dataset.

```
In [15]: np.random.seed(my_NIA)
```

```
In [16]: nan_atr=np.random.choice(list(X_train.columns), size=round(len(X_tr
         ain.columns)*0.1), replace=False)
```

```
In [17]: for i in range(0,len(nan_atr)):
             np.random.seed(i)
             nan_row=np.random.choice(X_train.index.tolist(), size=round(len
         (X_train.index)*0.1), replace=False)
             X_train.loc[nan_row,nan_atr[i]]=np.nan
```

```
/usr/local/lib/python3.7/site-packages/pandas/core/indexing.py:494
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pan
das-docs/stable/user_guide/indexing.html#returning-a-view-versus-a
-copy
  self.obj[item] = s
```

```
In [18]: for i in range(0,len(nan_atr)):
             np.random.seed(i)
             nan_row=np.random.choice(X_test.index.tolist(), size=round(len(
         X_test.index)*0.1), replace=False)
             X_test.loc[nan_row.tolist(),nan_atr[i]]=np.nan
```

Convertion to numpy array

```
In [19]: X_train=X_train.to_numpy()
         X_test=X_test.to_numpy()
```

Pipelines

SelectionKBest

Default parameters

```
In [ ]:
```

In [20]:
```
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsRegressor
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest, VarianceThreshol
d, f_regression
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV
from sklearn import metrics

imputer = SimpleImputer(strategy='median')
constant = VarianceThreshold(threshold=0.0) #Feature selector that
removes all low-variance features.
min_max_scaler = MinMaxScaler()
selector = SelectKBest(f_regression)
knn = KNeighborsRegressor()

selectkbest = Pipeline([
        ('impute', imputer),
        ('constant', constant),
        ('scaler', min_max_scaler),
        ('select', selector),
        ('knn_regression', knn)])

selectkbest = selectkbest.fit(X_train, y_train)

y_test_pred = selectkbest.predict(X_test)
print("\n The MSE is:\n",
      metrics.mean_squared_error(y_test_pred, y_test))
```

```
 The MSE is:
 11652549016879.31
```

Hyper-parameter tunning of the k number of features

In [21]:
```
param_grid = {'select__k': range(1,40,1)}

selectkbest_grid = GridSearchCV(selectkbest,
                          param_grid,
                          scoring='neg_mean_squared_error',
                          cv=tr_val_partition,
                          n_jobs=1, verbose=1)

selectkbest_grid = selectkbest_grid.fit(X_train, y_train)
y_test_pred = selectkbest_grid.predict(X_test)
print("\n The MSE is:\n",
      metrics.mean_squared_error(y_test_pred, y_test))
selectkbest_grid.get_params()
print(selectkbest_grid.best_params_)
```

```
Fitting 1 folds for each of 39 candidates, totalling 39 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.
[Parallel(n_jobs=1)]: Done  39 out of  39 | elapsed:    6.8s finis
hed

 The MSE is:
 11605113712871.627
{'select__k': 12}
```

Set the resulting k number after optimization, and look for the n_neighbors number that will give the best model

```
In [22]: selectkbest = selectkbest.set_params(**{'select__k':12})

         param_grid = {'knn_regression__n_neighbors': range(1,20,1)}

         selectkbest_grid = GridSearchCV(selectkbest,
                                   param_grid,
                                   scoring='neg_mean_squared_error',
                                   cv=tr_val_partition,
                                   n_jobs=1, verbose=1)

         selectkbest_grid = selectkbest_grid.fit(X_train, y_train)
         y_test_pred = selectkbest_grid.predict(X_test)
         print("\n The MSE is:\n",
               metrics.mean_squared_error(y_test_pred, y_test))
         print(selectkbest_grid.best_params_)
```

```
Fitting 1 folds for each of 19 candidates, totalling 19 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.

 The MSE is:
 11452919813548.143
{'knn_regression__n_neighbors': 6}

[Parallel(n_jobs=1)]: Done  19 out of  19 | elapsed:    3.3s finis
hed
```

PCA

PCA with default parameters

```
In [23]: from sklearn.decomposition import PCA

         imputer = SimpleImputer(strategy='median')
         constant = VarianceThreshold(threshold=0.0) #Feature selector that
         removes all low-variance features.
         min_max_scaler = MinMaxScaler()
         selector = PCA()
         knn = KNeighborsRegressor()

         pca = Pipeline([
                 ('impute', imputer),
                 ('constant', constant),
                 ('scaler', min_max_scaler),
                 ('select', selector),
                 ('knn_regression', knn)])

         pca = pca.fit(X_train, y_train)
         pca.get_params()

         y_test_pred = pca.predict(X_test)
         print("\n The MSE is:\n",
               metrics.mean_squared_error(y_test_pred, y_test))
```

```
 The MSE is:
 12195728226647.61
```

Hyper-parameter tunning of n_components

```
In [24]: param_grid = {'select__n_components': range(1,40,1)}

         pca_grid = GridSearchCV(pca,
                                 param_grid,
                                 scoring='neg_mean_squared_error',
                                 cv=tr_val_partition,
                                 n_jobs=-1, verbose=1)

         pca_grid = pca_grid.fit(X_train, y_train)
         y_test_pred = pca_grid.predict(X_test)
         print("\n The MSE is:\n",
               metrics.mean_squared_error(y_test_pred, y_test))
         print(pca_grid.best_params_)
```

```
Fitting 1 folds for each of 39 candidates, totalling 39 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  39 out of  39 | elapsed:    6.5s fini
shed

 The MSE is:
 11424299898401.908
{'select__n_components': 7}
```

Set parameter n_components, and find the optimal n_neighbors for the best model

```
In [25]: pca = pca.set_params(**{'select__n_components':7})

         param_grid = {'knn_regression__n_neighbors': range(1,20,1)}

         pca_grid = GridSearchCV(pca,
                                 param_grid,
                                 scoring='neg_mean_squared_error',
                                 cv=tr_val_partition,
                                 n_jobs=1, verbose=1)

         pca_grid = pca_grid.fit(X_train, y_train)
         y_test_pred = pca_grid.predict(X_test)
         print("\n The MSE is:\n",
               metrics.mean_squared_error(y_test_pred, y_test))
         print(pca_grid.best_params_)
```

```
Fitting 1 folds for each of 19 candidates, totalling 19 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.
[Parallel(n_jobs=1)]: Done  19 out of  19 | elapsed:    4.4s finis
hed

 The MSE is:
 10228944261569.63
{'knn_regression__n_neighbors': 19}
```

As the results show, the best model is the one with the PCA as the method for reducting the non important features of the dataset, this is the best result considering also the results from part I. The parameters resulted for the SelectKBest method were k=12 and n_neighbors=6, these means that the selecting method recognized 12 features that were representative from all the 300 variables, in contrast, the PCA identified only 7 components as the ones that had to be mantain and 19 neighbors.

Consequently, the PCA, gave a more accurate prediction and with less information, this is why it is consider as the best method of selection. Finally, the KNN gave the most accurate prediction of the amount of energy with both of the hyper-parameter tunnings. The summary of the MSEs results for the previous method are gathered in the next table:

Mean Square Error		
Machine learning selection methods	SelectKBest	PCA
KNN Default hyper-parameters	1,17E+13	1,22E+13
KNN selection hyper-parameter tunning	1,16E+13	1,14E+13
KNN n neighbors tunning	1,15E+13	1,02E+13
N of selected features/components	12	7
N of selected neighbors	6	19