Comparing the Performance using Solar Energy Production Data

Andres Mejia & Ignacio Almodovar

01/04/2022

Data preprocessing

We first split the training data into training and validation data. As the data is time ordered,the construction of the validation set must respect that ordering. The training set consists on the fist 3650 rows in the training set (roughly 10 years of data), and the validation set 730 that is a little more than 2 years of daily data. We will also only use the first 75 columns of the dataset as directed in the problem.

```
my_NIA =80762238
np.random.seed(my_NIA)
train = pd.read_pickle('/home/andres/AdprogSKlearn/traintestdata_pickle/trainst1ns16.pkl')
test = pd.read_pickle('/home/andres/AdprogSKlearn/traintestdata_pickle/testst1ns16.pkl')
train_target=train.loc[:,"energy"]
test_target=test.loc[:,"energy"]
train_train=train.iloc[:3650,0:75]
train_validation=train.iloc[3650:,0:75]
test1=test.iloc[:,0:75]
train_target_train=train_target.iloc[:3650]
train_target_validation=train_target.iloc[3650:]
```

Models using default parameters

We will first train the models using the default hyper-parameters.

Tree model

This model doesn't need scaling but a random state is added for reproducibility.

```
modelotree=DecisionTreeRegressor(random_state=1)
modelotree.fit(train_train,train_target_train)
```

```
tree_pred=modelotree.predict(train_validation)
```

KNN model

This model requieres scaling, it will be implemented in a pipeline.

```
pipeKNN = make_pipeline(StandardScaler(), KNeighborsRegressor())
pipeKNN.fit(train_train,train_target_train)
```

```
knn_pred=pipeKNN.predict(train_validation)
```

SVR model

As KNN this model also requieres scaling and is implemented in a pipeline.

```
pipeSVR = make_pipeline(StandardScaler(), SVR())
pipeSVR.fit(train_train,train_target_train)
```

```
SVR_pred=pipeSVR.predict(train_validation)
```

Now comparing the MAE of each model in the validation set we get:

```
metrics.mean_absolute_error(knn_pred,train_target_validation)
```

```
## 2588455.571506849
```

```
metrics.mean_absolute_error(tree_pred,train_target_validation)
```

```
## 3068353.430136986
```

```
metrics.mean_absolute_error(SVR_pred,train_target_validation)
```

```
## 6380505.183755267
```

The model that has the lowest mean absolute error in the validation set without any hyperparameter tuning is the k-nearest neighbors regression.

Tuning Hyperparameters

Tuning methodology

We will use the validation set to evaluate the hypertuning of parameters, instead of separating the sets we will use the function *PredefinedSplit* to mark the same rows as validation.

```
train_cv_index=np.zeros(train.shape[0])
train_cv_index[:3650] = -1
train_cv_index = PredefinedSplit(train_cv_index)
```

Tree Regressor

We define the search space as follows, we use *RandomizedSearchCV* to find the best hyperparameters. We also used absolute mean error as metric in the validation set in order to be consistant in the hyperparameter search.

```
max_features = ['log2', 'sqrt']
max_depth = [int(x) for x in np.linspace(1, 29, num = 12)]
min_samples_split = [2, 6, 10]
min_samples_leaf = [1, 3, 4]
random_grid = {
  'max_features': max_features,
  'max_depth': max_depth,
  'min_samples_split': min_samples_split,
  'min_samples_leaf': min_samples_leaf,
  'criterion':["mae"]
  }

rf_random = RandomizedSearchCV(estimator = modelotree,  param_distributions =
    random_grid, n_iter = 100, cv = train_cv_index, verbose=2, random_state=35,
    n_jobs = -1, scoring="neg_mean_absolute_error")
```

Comparing the results in the validation set we can see that the model with tuning has a lower absolute mean deviation.

```
rf_random.fit(train.iloc[:,0:75],train_target)
```

```
rf_random_pred=rf_random.predict(train_validation)
```

```
metrics.mean_absolute_error(tree_pred,train_target_validation)
```

```
## 3068353.430136986
```

```
metrics.mean_absolute_error(rf_random_pred,train_target_validation)
```

```
## 2343446.375342466
```

KNN regression.

For KNN there are a couple of changes, first we use *GridSearchCV* instead of *RandomizedSearchCV* to find the best hyperparameters. We also set the metric of the training to *p=1* so it uses the same metric in training.

```
n_neighbors=[3,4,5,6,7]
weight=["uniform","distance"]
algorithm=["ball_tree","kd_tree","brute"]
leaf_size=[10,20,30,40]
p=[1]


param_grid={
    "n_neighbors":n_neighbors,
    "algorithm":algorithm,
    "leaf_size":leaf_size,
    "p":p
    }


knn_estimator_cv=KNeighborsRegressor()
rs_knn = GridSearchCV(estimator =knn_estimator_cv, param_grid=param_grid,
                       cv = train_cv_index, verbose=2,
                      n_jobs = -1,scoring="neg_mean_absolute_error")
```

Comparing the results with and without optimization.

```
scaler1=StandardScaler().fit(train_train,train_target_train)
train_train_st=scaler1.transform(train_train)
train_st=scaler1.transform(train.iloc[:,0:75])
rs_knn.fit(train_st,train_target)
```

```
Knn_pred_2=rs_knn.predict(scaler1.transform(train_validation))
```

```
metrics.mean_absolute_error(knn_pred,train_target_validation)
```

```
## 2588455.571506849
```

```
metrics.mean_absolute_error(Knn_pred_2,train_target_validation)
```

```
## 2043643.6062622308
```

SVM Regression

We define the search space in a similar way, we use *RandomizedSearchCV* to find the best hyperparameters.

```
kernel=["linear","poly","rbf","sigmoid"]
degree=[1,2,3,4]
C=[x for x in np.linspace(start = 0.1, stop = 10, num = 10)]
shrinking=[True]

random_grid = {
   "kernel":kernel,
   "degree":degree,
   "C":C,
   "shrinking":shrinking}

scaler1=StandardScaler().fit(train_train,train_target_train)
train_train_st=scaler1.transform(train_train)
train_st=scaler1.transform(train.iloc[:,0:75])

SVR_estimatorS=SVR()
rs_svr = RandomizedSearchCV(estimator = SVR_estimatorS,
    param_distributions = random_grid,
    n_iter = 100,
    cv = train_cv_index,
    verbose=2, random_state=35, n_jobs = -1,scoring="neg_mean_absolute_error")
```

Again the optimized model behaves better in the validation set.

```
rs_svr.fit(train_st,train_target)
```

```
SVR_pred_2=rs_svr.predict(scaler1.transform(train_validation))
```

```
metrics.mean_absolute_error(SVR_pred,train_target_validation)
```

```
## 6380505.183755267
```

```
metrics.mean_absolute_error(SVR_pred_2,train_target_validation)
```

```
## 5674228.961291356
```

Model evaluation.

Now we will evaluate the models in the test set, note that we used the validation set in choosing the hyperparameters so in a way the training process got access to that data.

This code evaluates the model using test data and the optimized tree model.

```
tree_test=rf_random.predict(test.iloc[:,0:75])
v1=metrics.mean_absolute_error(tree_test,test_target)
```

The following table is generated using similar code for all models.

Method	MAE
KNN not optimized	2480348
KNN optimized	2312173
SVR not optimized	6383006
SVR optimized	5727618
Tree not optimized	2904819
Tree optimized	2571934

Final model training

The best model in the test set is the optimized KNN, we will train the model with all available data.

```
totaldata=train.append(test)
final_model_params=rs_knn.best_params_
finalKNN=make_pipeline(StandardScaler(), KNeighborsRegressor(**final_model_params))
pipeKNN.fit(totaldata.iloc[:,0:75],totaldata.loc[:,"energy"])
```

```
## Pipeline(steps=[('standardscaler', StandardScaler()),
##                 ('kneighborsregressor', KNeighborsRegressor())])
```