PREDICTING WIND ENERGY PRODUCTION WITH SCIKIT-LEARN

ASSIGNMENT 2 - ADVANCED PROGRAMMING

Errasti Domínguez, Nuria and Levenfeld Sabau, Gabriela

Master in Statistics for Data Science

Universidad Carlos III de Madrid

January 2024

Contents

1 Introduction **1**

2 Exploratory Data Analysis (EDA) **2**

2.1 Question 2. 2

3 Machine learning algorithms **5**

3.1 Question 3. Split into train and test 5

3.2 Question 4. Default hyper-parameters: Trees and KNN 6

3.3 Question 5. Hyper-parameter tuning: Trees and KNN 9

3.3.1 Random-search 9

3.3.2 Bayesian-search (question 1-a) 12

3.3.3 Results summary 14

3.4 Question 6. Best method 14

3.4.1 Estimation of the error at the competition 14

3.4.2 Final model 15

3.5 Question 7. Feature selection for KNN 16

4 Conclusions **20**

1 Introduction

Before starting the project, we will give a brief description of its organization and structure. For the assignment, we created two main folders.

Folder: `src`

It contains the project's code written in Python. The following scripts are located in this directory:

1. `EDA.py`; where we explore the dataset and perform the simplified Exploratory Data Analysis (EDA).
2. `sketch.py`; where we perform the different tasks related to the models and how to train them, as well as computing the final predictions.
3. `utils.py`; set of custom utility functions designed to be reused during the task.
4. `knn_bayesian_search.py`; script to store functions related to bayesian-search using the Optuna library.

Folder: `results`

It contains the different outputs generated during the assignment. Here, the following files can be found:

- models_summary.csv; summary with the results of the different models which include the **execution time of training process** of all models. However, this values have not being added on this report file. (question 5)
- final_model.joblib; (question 6).
- predictions_wind_competition.csv; file for saving the predictions for the competition dataset (question 6).
- KNNPredictBO.db; database used to store the bayesian-search process.
- trials_info.csv; file containing all combination attempts during the bayesian search.

Lastly, it's important to mention that in the report, we have only included the code we consider essential for our explanations. For more detailed information on the code, please refer to the accompanying scripts.

2 Exploratory Data Analysis (EDA)

After importing the dataset, we found that it consisted of 4748 observations and 555 variables. The features include information on different qualities such as time, wind, temperature... for many locations (25 differents locations). All variables are numerical, being 551 real numbers and 4 integer numbers (which makes sense since they correspond to hour, day, month and year variables). Our dataset is quite large and complicated considering it has 555 variables, so before starting with the modelling problem we must do an exploratory data analysis.

2.1 Question 2.

After checking for constant columns or duplicated data, none were found. However, quite a large number of NA's were found (Total NA in the dataset: 326132) which sum up to be around a 12% of the data. The only columns without NA's are energy, year, month, day and hour. We can also identify the missing values graphically. The following are the heatmap of total NA's in the dataset and the heatmap of NA's in the variables at the Sotavento point (variables that finish with '13').

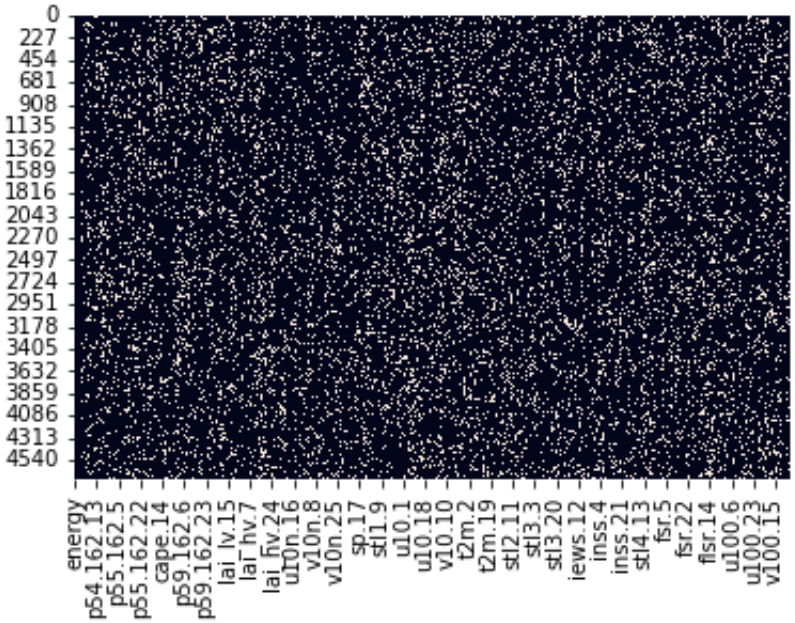

Figure 1: Missing values heatmap of the whole dataset

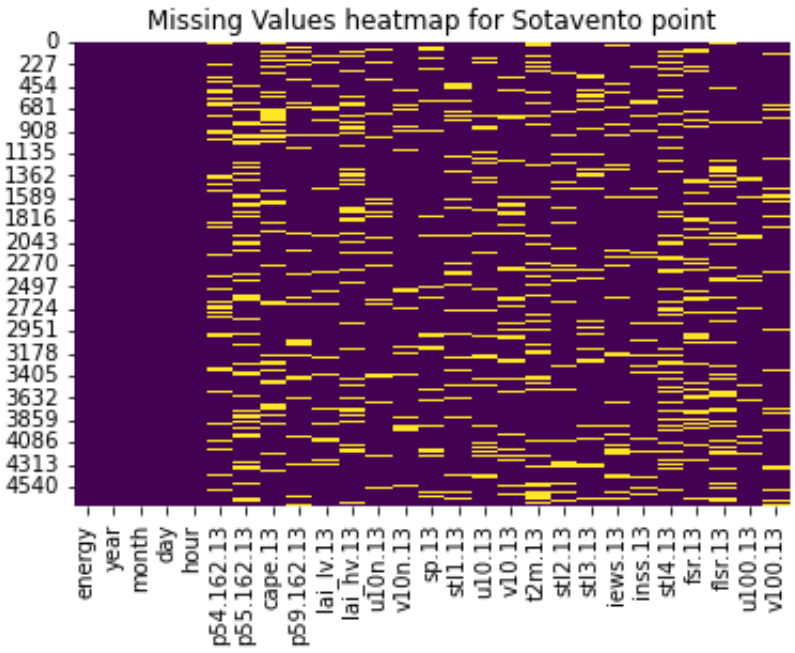

Figure 2: Missing values heatmap in the Sotavento point

The target variable is energy which is continuous, therefore we have a regression problem. In order to understand better the problem, we plotted the energy variable and observed that the year 2008 presents weird results (during this year only 178 instances were recorded comparing to the average of 1200 in other years).

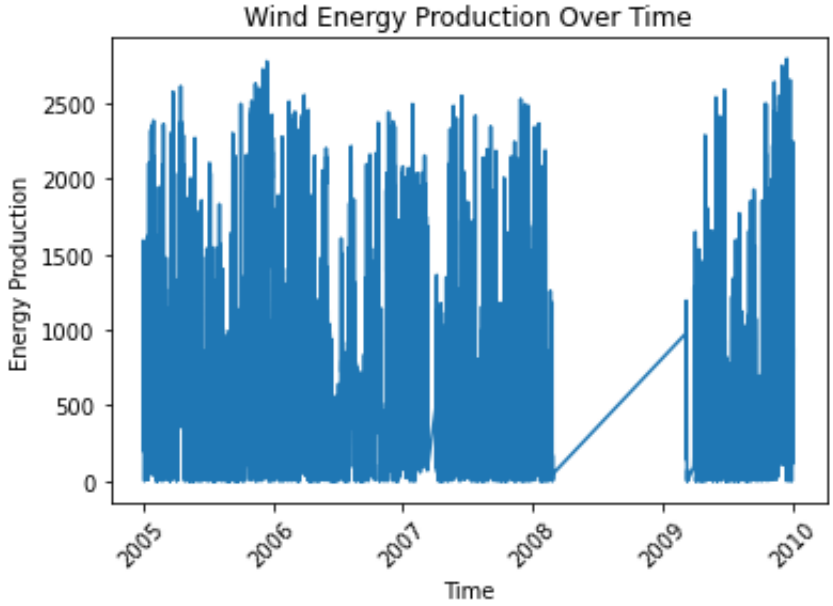

Figure 3: Wind energy production over time

Thanks to the exploratory data analysis we have found that there are many missing values which we will need to impute later on in order to work with KNN and tree models. We have also found that the year 2008 has little representation in the dataset, therefore when separating the training set into train and validate we must be careful and take this into account.

3 Machine learning algorithms

3.1 Question 3. Split into train and test

When splitting the train dataset, we must take into account that our data is non i.i.d (independent and identically distributed) due to its temporal order. Because of this reason, we must maintain the temporal order during the split. This can be done separating the data by years. To make the corrects decision of which years belong to each dataset, we must take a look at how many observations there are per year (see table below).

Year	**Number of observations**
2005	1256
2006	1272
2007	1121
2008	178
2009	921

Table 1: Number of observations per year.

Based on the distribution, we decided to use data from 2005, 2006, 2007 and 2008 for the train_train dataset, reserving the year 2009 for the train_validation dataset. This partition leads us into an 80% portion for training and a 20% portion for validation, aligning with standard practice to ensure representative datasets for consistent decision-making.
Last decision to make is which metric will be used to evaluate the different models. Given that we are dealing with a regression problem, specifically predicting wind energy production, we choose three metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Coefficient of Determination (R^2).

- **Mean Absolute Error** (MAE); This metric provides the average value of the total absolute differences between the predicted and actual values. Lower values indicate better performance, approaching zero.
- **Root Mean Squared Error** (RMSE); RMSE is the square root of the average of the squared differences between the model predictions and the actual values. Also, it penalizes large errors more than smaller errors. The closer to zero, the better the model's performance is.
- **Coefficient of Determination** (R^2); This measure tells us how well a regression model fits the actual data. It quantifies the degree to which the variance in the

dependent variable is predictable from the independent variables. Values closer to one indicate a better performance.

Finally, we include the essential code for answering this question.

Listing 1: split_data function

```
def split_data(data, years_train_train):
    """ Split the train data into train_train (80%) and train_val (20%) datasets. """

    # For train_train we take 2005, 2006, 2007 and 2008.
    train_train = data[data['year'].isin(years_train_train)]
    X_train = train_train.drop(columns='energy')
    y_train = train_train['energy'].values

    # For train_validation we take 2009.
    train_val = data[~data['year'].isin(years_train_train)]
    X_val = train_val.drop(columns='energy')
    y_val = train_val['energy'].values

    return X_train, y_train, X_val, y_val
```

Listing 2: eval_metrics_model function

```
def eval_metrics_model(y_val, y_val_pred):
    """ Function to compute and print performance metrics: MAE, RMSE and R-squared. """

    mae = metrics.mean_absolute_error(y_val, y_val_pred)
    rmse = metrics.mean_squared_error(y_val, y_val_pred, squared=False)
    r2 = metrics.r2_score(y_val, y_val_pred)

    print(f'Mean Absolute Error (MAE): {mae}')
    print(f'Root Mean Squared Error (RMSE): {rmse}')
    print(f'R-squared: {r2}')

    return {'MAE': mae, 'RMSE': rmse, 'R^2': r2}
```

3.2 Question 4. Default hyper-parameters: Trees and KNN

The first task consists on creating several models with default hyper-parameters. As an important consideration, we have to remember that the models we are going to work with can not deal with missing values, so we need to handle them.

KNN Model

For the KNN model, we employ the same kind of imputation for completing the missing

values. However, we explore three different types of standardization. The following step is to create different pipelines, then we train (fit) the models with the train dataset. After that, we use these models to predict the values for the validation set. And the final step involves evaluating them using the predefined metrics.

Listing 3: Default hyper-parameter: KNN workflow

```
# KNN model
imputer_knn = KNNImputer() # Imputation transformer for completing missing values
knn = KNeighborsRegressor()

# Pipeline for KNN with Standard Scaler
reg_knn_std = Pipeline([
    ('imputation', imputer_knn),
    ('standarization', StandardScaler()),
    ('knn', knn)
    ])

# Pipeline for KNN with Robust Scaler
reg_knn_robust = Pipeline([
    ('imputation', imputer_knn),
    ('standarization', RobustScaler()),
    ('knn', knn)
    ])

# Pipeline for KNN with MinMax Scaler
reg_knn_minmax = Pipeline([
    ('imputation', imputer_knn),
    ('standarization', MinMaxScaler()),
    ('knn', knn)
    ])

# Fit and evaluate KNN with Standard Scaler
reg_knn_std.fit(X_train, y_train)
y_val_pred_std = reg_knn_std.predict(X_val)
metrics_knn_std = eval_metrics_model(y_val, y_val_pred_std)

# Fit and evaluate KNN with Robust Scaler
reg_knn_robust.fit(X_train, y_train)
y_val_pred_robust = reg_knn_robust.predict(X_val)
metrics_knn_robust = eval_metrics_model(y_val, y_val_pred_robust)

# Fit and evaluate KNN with MinMax Scaler
reg_knn_minmax.fit(X_train, y_train)
```

```
y_val_pred_minmax = reg_knn_minmax.predict(X_val)
metrics_knn_minmax = eval_metrics_model(y_val, y_val_pred_minmax)

# Plot for the results - Standard Scaler (with better performance)
plot_predictions(y_val, y_val_pred_std, 'KNN (Standard Scaler)')
```

The last line of the provided code computes a graphical overview of how closely the predicted values are with the actual values for a specific model. In this case, we plot the **KNN with Standard Scaler** standarization model, which produces the lowest error. Thus, it performances better than the rest of the models.

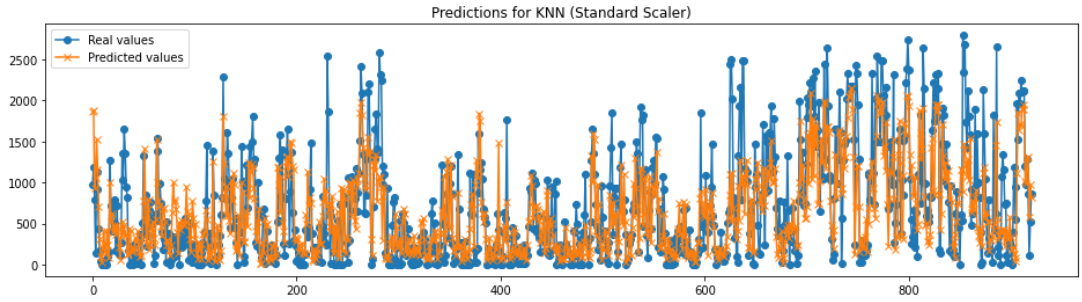

Figure 4: Predictions for KNN (Standar Scaler) default hyper-parameters

Tree Model

The workflow for the Tree model is kind of the same as the one described above for the KNN model, with the only distinction being the absence of standardization because it is not necessary.

Listing 4: Default hyper-parameter: Tree workflow

```
# Tree model
SEED = 100507449
np.random.seed(SEED)

# We define the type of training method (nothing happens yet)
tree_model = tree.DecisionTreeRegressor(random_state=SEED)
imputer_tree = SimpleImputer(strategy='mean') # Imputation transformer for completing
    missing values

# Pipeline for Tree
reg_tree = Pipeline([
    ('imputation', imputer_tree),
    ('tree', tree_model)
    ])
```

```
reg_tree.fit(X_train, y_train) # Now, we train (fit) the method on the train dataset
y_val_pred_tree = reg_tree.predict(X_val) # We use the model to predict on the
    validate set
metrics_tree = eval_metrics_model(y_val, y_val_pred_tree) # Evaluate the model

# Plot for the results
plot_predictions(y_val, y_val_pred_tree, 'Tree')
```

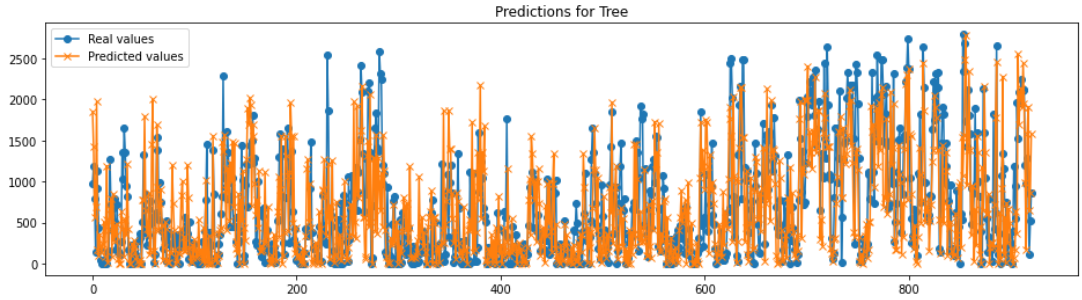

Figure 5: Predictions for Tree default hyper-parameters

Best model performance

Among all the models tested with default hyper-parameters, the **KNN with Standard Scaler** demonstrates the lowest error, indicating a better performance.

3.3 Question 5. Hyper-parameter tuning: Trees and KNN

The idea behind hyper-parameter tuning is to select the best parameters in order to improve the model's performance. Several techniques can be used for this search; among them we can find: grid-search, random-search or bayesian-search. However, for this section, we will not use *grid-search* because it is computationally expensive and time-consuming, since it tries all possible combinations.

The first section for answering question 5 involves a ***random-search*** for both KNN and Tree models, as shown during lectures, it randomly tests only some of the possible combinations. After that, we found interesting to implement a ***bayesian-search*** using the Optuna library. Finally, we will provide a summary with the results of the different alternative tested over the project.

3.3.1 Random-search

Some workflow is defined similarly for both KNN and Tree models. First, we establish the **search space**, specifying the method and its allowed ranges of values. Here, we can include categorical as well as numeric variables. As before, we create a pipeline to preprocess

the data (imputation, standardization if necessary and applying the corresponding model). During the random-search, we set the number of trials to 20, randomly sampled, and use MAE as the strategy to evaluate model performance. Also, we define a fixed train/validation grid-search with no shuffling, where -1 represents training and 0 means validation. The last step involves training the model with the parameters selected for the random grid-search, making predictions on the validation set and storing the performance metrics for future comparisons between models.

Listing 5: Defining a fixed train/validation grid-search

```
# CODE Used for both models: KNN & Tree
validation_indices = np.zeros(X_train.shape[0])
validation_indices[:round(2/3*X_train.shape[0])] = -1
tr_val_partition = PredefinedSplit(validation_indices)
```

KNN Model

Listing 6: HPO with random-search: KNN workflow

```
# Defining the method (KNN) with pipeline
reg_knn_hpo = Pipeline([
    ('imputation', imputer_knn),
    ('standarization', StandardScaler()),
    ('knn', knn)
    ])

# Defining the Search space
param_grid_knn = {'knn__n_neighbors': sp_randint(2,16,2),
                  'knn__weights': ['uniform', 'distance'],
                  'knn__leaf_size': sp_randint(10,50,2)}

reg_knn_grid = RandomizedSearchCV(reg_knn_hpo,
                          param_distributions=param_grid_knn,
                          n_iter=20,
                          scoring='neg_mean_absolute_error',
                          cv=tr_val_partition,
                          n_jobs=4, verbose=1)

reg_knn_grid.fit(X_train, y_train) # Training the model with the grid search
y_val_pred_hpo_knn = reg_knn_grid.predict(X_val) # Making predictions on the validate
    set
metrics_hpo_knn = eval_metrics_model(y_val, y_val_pred_hpo_knn) # Evaluate the model

# Plot the results
```

```
plot_predictions(y_val, y_val_pred_hpo_knn, 'HPO KNN')

# The best hyper parameter values (and their scores) can be accessed
print(f'Best hyper-parameters: {reg_knn_grid.best_params_} and inner evaluation:
    {reg_knn_grid.best_score_}')
```

Best hyper-parameters: 'knn__leaf_size': 25, 'knn__n_neighbors': 15, 'knn__weights': 'distance' and inner evaluation: -323.60454579041004

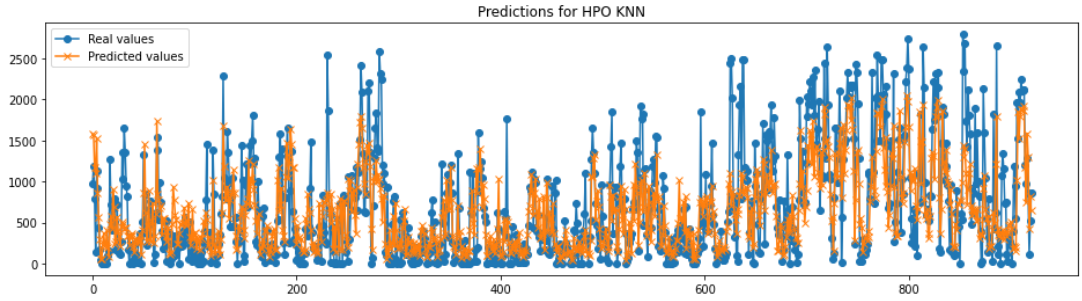

Figure 6: Predictions for KNN HPO with random-search

Tree Model

Listing 7: HPO with random-search: Tree workflow

```
# Defining the method with pipeline
reg_tree_hpo = Pipeline([
    ('imputation', imputer_tree),
    ('tree', tree_model)
    ])

# Defining the Search space
param_grid_tree = {'tree__max_depth': sp_randint(2,16,2),
                   'tree__min_samples_split': sp_randint(2,34),
                   'tree__min_samples_leaf': sp_randint(1,30,5)}

reg_tree_grid = RandomizedSearchCV(reg_tree_hpo,
                                   param_distributions=param_grid_tree,
                                   n_iter=20,
                                   scoring='neg_mean_absolute_error',
                                   cv=tr_val_partition,
                                   n_jobs=3, verbose=1)

reg_tree_grid.fit(X_train, y_train) # Training the model with the grid search
```

```
y_val_pred_hpo_tree = reg_tree_grid.predict(X_val) # Making predictions on the
    validate set
metrics_hpo_tree = eval_metrics_model(y_val, y_val_pred_hpo_tree) # Evaluate the model

# Plot for the results
plot_predictions(y_val, y_val_pred_hpo_tree, 'HPO Tree')

# The best hyper parameter values (and their scores) can be accessed
print(f'Best hyper-parameters: {reg_tree_grid.best_params_} and inner evaluation:
    {reg_tree_grid.best_score_}')
```

Best hyper-parameters: 'tree__max_depth': 9, 'tree__min_samples_leaf': 15, 'tree__min_samples_split': 10 and inner evaluation: -316.36588459832495

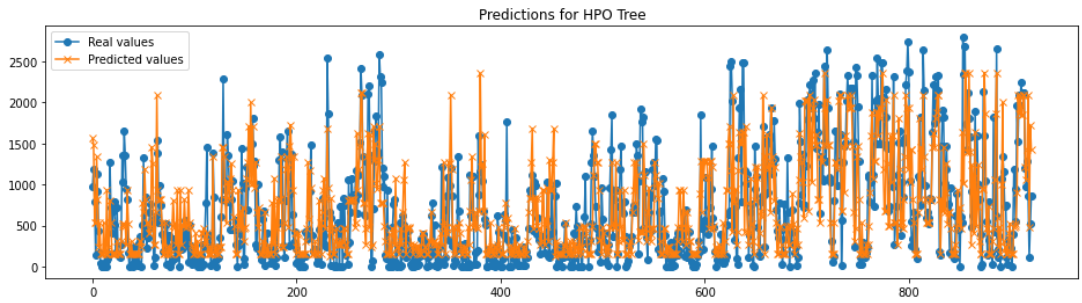

Figure 7: Predictions for Tree HPO with random-search

3.3.2 Bayesian-search (question 1-a)

The second section focuses on implementing the bayesian-search specifically for the KNN model. This decision was reached after observing that, during default parameter and hyper-parameter tuning, the KNN model always outperformed the Tree model.
In general terms, bayesian optimization is a type of search that builds a statistical model to predict which combination of hyper-parameters is likely to generate a good performance. After, the result obtained guides the selection of the most promising next configuration. The Optuna library allows us to set upper and lower bounds to explore ranges of both numerical and categorical values.
We have decided to store this process, where 60 combinations are evaluated, in a database called "KNNPredictBO.db.". On the other hand, it is worth mentioning that the duration of each combination has been processed in approximately 1 minute. Also, we use the Mean Absolute Error (MAE) as the strategy to evaluate model performance. The search has been

applied both to the preprocessing step (standardization) as well as to the hyper-parameters of the KNN model.

The specific code related to this section can be found in *src/knn_bayesian_search.py*. Furtheremore, a file has been created to store all the trials' information related with the bayesian-search, which is located at *results/trials_info.csv*.

Listing 8: KNN HPO with bayesian-search

```
# Create Optuna study
study_knn = optuna.create_study(study_name='AP_Task2',
                                direction='minimize',
                                storage = 'sqlite:///KNNPredictBO.db',
                                load_if_exists=True)
# Bayesian search
n_trials=60
best_params_bo = param_search_bo((X_train, y_train), (X_val, y_val), study_knn,
    n_trials)
best_mae_bo = study_knn.best_value

print(f'Best hyper-parameters: {study_knn.best_params}')
```

Best hyper-parameters: 'scalers': 'standard', 'n_neighbors': 12, 'weights': 'distance', 'algorithm': 'kd_tree', 'leaf_size': 20

3.3.3 Results summary

Table 2: Model evaluation results

Model	Search	Standardization	Best Hyperparameters	Validation MAE	Validation R^2	Validation RMSE
KNN	None	Standard	Default	311.92	0.57	435.64
KNN	None	Robust	Default	324.54	0.54	452.85
KNN	None	MinMax	Default	334.32	0.51	466.01
Tree	None	None	Default	386.63	0.37	530.88
KNN	Random-search	Standard	{'knn__leaf_size':25, 'knn__n_neighbors':15, 'knn__weights':'distance'}	308.43	0.59	425.48
Tree	Random-search	None	{'tree__max_depth':9, 'tree__min_samples_leaf':15, 'tree__min_samples_split':10}	333.44	0.51	465.22
KNN	Bayesian-search	Standard	{'n_neighbors':12, 'weights': 'distance', 'algorithm':'kd_tree', 'leaf_size':20}	303.63	NaN	NaN

From this summary, we can make some conclusions:

- The KNN model with bayesian-search achieved the lowest MAE (303.63), indicating best performance compared to the other configurations.
- The default configurations for both the KNN and Tree models resulted in higher errors compared to the tuning models, which emphasizes the importance of including hyper-parameter tuning.

3.4 Question 6. Best method

3.4.1 Estimation of the error at the competition

As we mentioned before, the best method of the evaluated ones is the **KNN model with bayesian-search**, which produced the lowest error on the validation test. From this model, we compute the mean absolute error (MAE) which gives us an idea of how the model is going to work with unseen data. Its value is 303.63, suggesting that for a new instance, our model is expected to predict the response (in our case, the energy variable) with an absolute difference of 303.63 units from the actual values. In other words, the prediction can be either 303.63 units higher or 303.63 units lower than the real value.

3.4.2 Final model

Best method

Listing 9: Code for searching the best method

```
# Look for the best model
df = pd.read_csv('results/models_summary.csv')
best_model = df.loc[df['Validation MAE'].idxmin()]
print(f'The best model is {best_model["Model"]} with {best_model["Search"]}.')
```

The best model is KNN with Bayesian-search.
Parameter selected:
'scalers': 'standard', 'n_neighbors': 12, 'weights': 'distance', 'algorithm': 'kd_tree', 'leaf_size': 20

Training the final model and making predictions on the competition dataset

Listing 10: Training and making new predictions

```
# best_params_bo has already been defined and it carries out the parameters of the HPO
    with bayesian search
final_model = Pipeline([
    ('imputation', imputer_knn),
    ('standarization', best_scaler),
    ('knn', KNeighborsRegressor(n_neighbors=best_params_bo['n_neighbors'],
                                weights=best_params_bo['weights'],
                                algorithm=best_params_bo['algorithm'],
                                leaf_size=best_params_bo['leaf_size']))
    ])

start_time = time.time()
# In order to get the final model we need to train using the entire dataset (X,y)
final_model.fit(X, y)
end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")

# Now, we can use the final_model to make predictions on new data
wind_comp = load_data('data/wind_competition.csv.gzip') # Load new data
pred_new = final_model.predict(wind_comp)
```

Execution time: 37.410258531570435 seconds

Save final model and the competition predictions on files

The predictions generated by the best model have been stored in a new file named *predictions_wind_competition.csv*, specifically in a new column labeled "Predictions". Additionally, the final model can be found in the file named *final_model.joblib*.

Listing 11: Saving files

```
wind_comp['Predictions'] = pred_new
# Save predictions for the competition dataset
wind_comp.to_csv('results/predictions_wind_competition.csv', index=False)

# Save the final_model
dump(final_model, 'results/final_model.joblib')
```

3.5 Question 7. Feature selection for KNN

The goal of this final section is to identify the attributes of the dataset which influence more when predicting the response variable. With an initial dataset of 555 features, managing this complexity can become challenging. Therefore, a good approach is to do a feature selection in order to reduce the number of variables to work with. In this assignment, we evaluated the attributes with a linear correlation threshold, working with the SelectKBest() function.

Even thought we did a hyper-parameter tuning previously, the question 7 specifies setting the optimal value only for the n_neighbors parameter, leaving the remaining parameters at their default values. The selection part is included into the pipeline, as an additional pre-processing step. When defining the search space, we emphasize the decision made on determining the range for k, which represents the number of top features to select. Since we are using grid-search, this lead us into a 72 candidate (combinations) during the search. Finally, we display the results.

Listing 12: Feature selection for KNN code

```
# Set the global configuration to keep DataFrame structure in transformers' output
set_config(display='diagram', transform_output='pandas')

# Create the pipeline for defining the method KNN
# best_scaler; param getting for the bayesian-search
selector = SelectKBest()
knn_fs = KNeighborsRegressor(n_neighbors=best_params_bo['n_neighbors'])
```

```
reg_knn_fs = Pipeline([
    ('imputation', imputer_knn),
    ('standarization', best_scaler),
    ('select', selector),
    ('knn', knn_fs)
    ])

# Defining hyper-parameter space
param_grid_fs = {'select__k': list(range(4, 40, 1)),
                 'select__score_func': [f_regression, mutual_info_regression]}

reg_fs_grid = GridSearchCV(reg_knn_fs,
                           param_grid=param_grid_fs,
                           scoring='neg_mean_absolute_error',
                           cv=tr_val_partition,
                           verbose=1)

reg_fs_grid.fit(X_train, y_train) # Training the model with the grid search
y_val_pred_fs = reg_fs_grid.predict(X_val) # Making predictions on the validation set
metrics_fs = eval_metrics_model(y_val, y_val_pred_fs) # Evaluate the model

# Get the best score
best_score_fs = reg_fs_grid.best_score_
best_params_fs = reg_fs_grid.best_params_
print(f'Best score (negative mean absolute error): {best_score_fs}')
print(f'Best params: {best_params_fs}')

# Get the selected feature scores and names
selected_feature_mask = reg_fs_grid.best_estimator_.named_steps['select'].get_support()
feature_scores =
    reg_fs_grid.best_estimator_.named_steps['select'].scores_[selected_feature_mask]
selected_feature_names = X_train.columns[selected_feature_mask]
print("Selected features and scores:")
for elem in zip(selected_feature_names, feature_scores):
    print(elem)

# Sort the selected features based on scores
sorted_features = sorted(zip(selected_feature_names, feature_scores), key=lambda x:
    x[1], reverse=True)
print("Selected features and scores sorted")
for elem in sorted_features:
    print(elem)
```

Display results

Best score (negative mean absolute error): -312.56939393939393
Best params: 'select__k': 38, 'select__score_func': <function mutual_info_regression at 0x0000012E77085260>

Selected features and scores sorted

('iews.13', 0.3604729003856013) ('iews.11', 0.35889859734066043) ('iews.18', 0.3549088726234917) ('iews.19', 0.3512969275097424) ('u100.11', 0.34934451807077327) ('iews.21', 0.3488835154079215) ('iews.20', 0.3458060802762546) ('iews.17', 0.34312853480968375) ('iews.14', 0.3429775019809469) ('u100.24', 0.33796760538495185) ('iews.2', 0.33729167101627056) ('iews.6', 0.3363754656035365) ('iews.12', 0.3325065753536913) ('iews.4', 0.32823919258907086) ('u100.1', 0.3270458739960356) ('u100.22', 0.3259160731038957) ('u100.8', 0.3254089378644114) ('u100.13', 0.32497779902815616) ('iews.23', 0.32450193380428516) ('iews.7', 0.32417556825423066) ('iews.8', 0.3240735222236726) ('iews.24', 0.32131919990438185) ('iews.16', 0.3212633698952141) ('u100.25', 0.32113589925495756) ('u100.15', 0.320814427528922) ('u100.7', 0.32076358606580246) ('u100.17', 0.3160639133789589) ('iews.5', 0.31568207310693364) ('u100.4', 0.3144042312912916) ('u100.9', 0.31038468957811727) ('iews.25', 0.31002785131154553) ('iews.22', 0.30787721928569667) ('iews.10', 0.30603602931167195) ('u100.18', 0.3053723866478162) ('u100.6', 0.30497054848279426) ('u100.3', 0.3047724647884582) ('u100.10', 0.30289710683331705) ('u100.5', 0.3012738578521068)

Interpretation of the results

The initial number of features of the dataset was 555. During the grid-search we set the parameter that selects the important features to choose up to 40 variables. After computing the grid-search the best k number of features was 38. When looking at the 38 best features, we quickly realize that all of them are related to the $iews$ (instantaneous eastward turbulent surface stress) and $u100$ (100 metre U wind component) variables, specially being the most important the $iews$ variable at the Sotavento location. However, the next most important features are not the ones measured at the Sotavento point. This makes sense, because these two attributes are directly related to wind properties. Finally, if we take a look at how many selected features there are per location point, we realise all of them have a representation.

It looks like most of the points affect the prediction at the Sotavento point in the same way, having 50% of the points 1 relevant feature and the other 50% of the points 2 selected features. Furthermore, there is no a clear pattern of a specific area with more influence than other.

Lastly, we can conclude that this new model does not improve over the previously tested. There are several models which perform better.

Remark: We also tried to set the k to be up to 80 features, the result was that the best number of features is 78. The next best features were those related to the variables $u10$ and $u10n$. However, since the complexity of computing this was much larger we decided to keep the search of features up to 40. Furthermore, we also asked to choose k from a list starting at 4 that went up to 200 with a step of 4. In this case, the best k was 100. Therefore, we can conclude that the best number of features is indeed much smaller than the original number.

4 Conclusions

We started this project with a big dataset that would cause problems when starting to work with it. The number of variables was huge (555) and most of them included missing values. In order to work with them, the missing values would need to be implemented and we did so. Next, we started by trying out different machine learning algorithms to model our regression problem. The methods used already have some default parameters in Python, however sometimes they are not optimal for one problem. Therefore, our next step was to try hyper-parameter tuning in order to choose the best parameters for our specific problem. The best method was the **KNN model with bayesian-search** with 12 being the number of neighbors. Furthermore, we also carried out hyper-parameter tuning in order to choose the best number of features to reduce the complexity of our problem. We obtained that the best number of variables was indeed much smaller than the original, and most of them were related to $iews$ and $u100$.