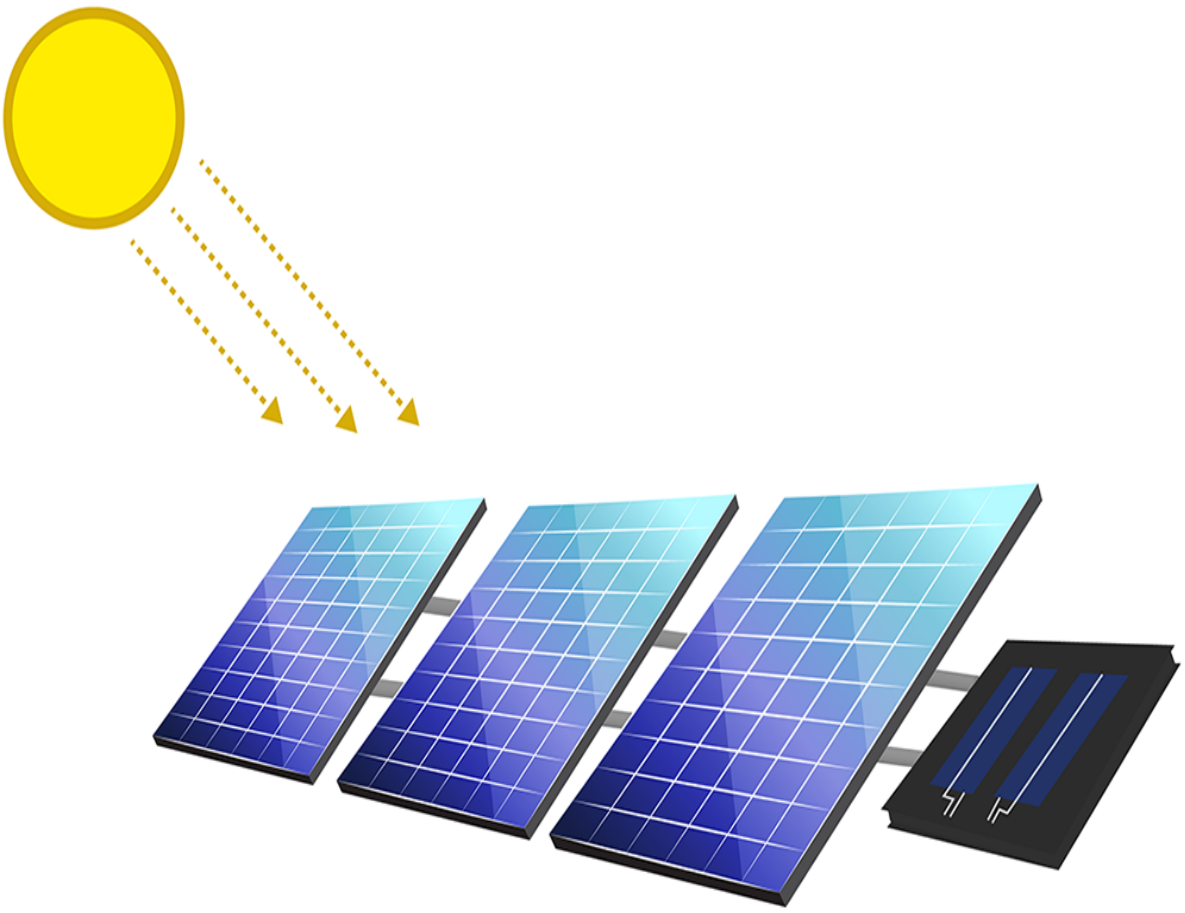
# PREDICTING SOLAR ENERGY PRODUCTION WITH SCIKIT-LEARN PART I



JAVIER FONG GUZMAN
MENGTING CHEN

We are going to use meteorological variables forecasted by GFS as input attributes to a machine learning model that is able to estimate how much solar energy is going to be produced at one of the solar plants in Oklahoma.

1) Make the results are reproducible using np.random.seed:

```
my_NIA = 100466870
np.random.seed(my_NIA)
```

2) Read the solar dataset into a Pandas dataframe:

We have two files, the train dataset contains data from 1994-2007 and the test dataset, for 2008-2009. We have a total of 15 meteorological variables, forecasted for 5 different times of the day, at 16 locations (blue points) around the solar plant of interest (red point). The last column (energy) is the response variable.

```
os.chdir("/Users/javi.fong/Documents/MS DataScience/2 Cuatrimestre/Advance Programming/Advance Programming - Assignment 2")
train = pd.read_pickle("traintestdata_pickle/trainst1ns16.pkl")
test = pd.read_pickle("traintestdata_pickle/teststst1ns16.pkl")
```

3) Select only the closest “blue point” to the corresponding red point.
Selecting the first 75 columns:

```
#Choose closest point variables
train_x = train.iloc[:,:75]
train_y = train.energy

test_x = test.iloc[:,:75]
test_y = test.energy
```

In this section we will use three different machine learning algorithms: KNN, SVMs and Regression Trees with and without hyper-parameter tuning.

To start with model selection, we split the train data into train_train and train_validation, and the validation set will be used to compare all six approaches.

**MODEL SELECTION**

- WITH DEFAULT HYPER-PARAMETERS

Algorithms such as KNN and SVMs require scaling, we will use Pipeline for this purpose. To fit the model we use the train_train dataset and for prediction the train_validation dataset.

KNN with default hyper-parameter:

```
### KNN with default parameter
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn import neighbors

scaler = StandardScaler()
knn = neighbors.KNeighborsRegressor()
classif_knn = Pipeline([
    ('standarization', scaler),
    ('knn', knn)
    ])
classif_knn.fit(train_train, train_train_Y)
y_hat_knn=classif_knn.predict(train_validation)
print("KNN with Default HP:" ,sk.metrics.mean_absolute_error(train_validation_Y, y_hat_knn))
```

SVMs with default hyper-parameters:

```
svm = sk.svm.SVC()
classif_svm = Pipeline([
    ('standarization', scaler),
    ('svm', svm)
    ])
classif_svm.fit(train_train, train_train_Y)
y_hat_svm=classif_svm.predict(train_validation)
print("SVM with Default HP:", sk.metrics.mean_absolute_error(train_validation_Y, y_hat_svm))
```

Regression Tree with default hyper-parameter:

```
#### Regression Tree with default parameter
from sklearn import tree
classif_rt= tree.DecisionTreeRegressor()
classif_rt.fit(train_train, train_train_Y)
y_hat_rt=classif_rt.predict(train_validation)
print("Decission Tree with Default HP:",sk.metrics.mean_absolute_error(train_validation_Y, y_hat_rt))
```

The results of the three algorithms are summarised in the table below:

| Algorithm | Mean absolute error |
|---|---|
| KNN | 2.588.455,571506849 |
| SVM | 6.920.271,25479452 |
| Regression Tree | 3.078.653,835616438 |

Without hyper-parameter tuning the best algorithm using mean absolute error as the metric for evaluation is KNN.

- WITH HYPER-PARAMETER TUNING

We will use hyper-parameter tuning with RandomizedSearch for Regression Trees and SVMd, and for KNN the GridSearch to optimize just the number of neighbors. The evaluation of the best hyper-parameter will be done on the validation set.

KNN with hyper-parameter tuning(number of neighbors):

We start defining the search space for GridSearchCV, in this case the number of neighbors. GridSearchCV is going to evaluate all values of neighbors defined in the search space using KNN and give us the best according to mean absolute error.

```
param_grid_knn= {'n_neighbors': list(range(1,16,1))}
knn_grid= GridSearchCV(
    knn
    , param_grid_knn
    , scoring='neg_mean_absolute_error'
    , cv= tr_val_partition
    , verbose = 1
    )
knn_grid.fit(scaled_all_train_x, cp_train_Y) ## use scale data
print(f'Best hyper-parameters: {knn_grid.best_params_} and inner evaluation: {knn_grid.best_score_}')
```

SVM with hyper-parameter tuning:

In this case for the value of search space we have used statistical distribution out of which values can be sampled.

```
budget = 20
svm_grid = RandomizedSearchCV(
    svm
    , param_grid_svm
    , scoring = 'neg_mean_absolute_error'
    , cv = tr_val_partition
    , n_iter = budget
    , n_jobs = 1
    , verbose = 1
    )
svm_grid.fit(scaled_all_train_x, cp_train_Y) ### Use scale data
print(f'Best hyper-parameters: {svm_grid.best_params_} and inner evaluation: {svm_grid.best_score_}')
```

Regression Tree with hyper-parameter tuning:

```
param_grid_rt = {
    "min_samples_split": list(range(2,16,2))
    , "max_depth": list(range(2,16,2))
    , "min_samples_leaf": list(range(2,16,2))
    , "max_leaf_nodes": list(range(2,16,2))
    }

budget= 20
rt_grid = RandomizedSearchCV(
    classif_rt
    , param_grid_rt
    , scoring = 'neg_mean_absolute_error'
    , cv=tr_val_partition
    , n_iter=budget
    , n_jobs=1
    , verbose =1 )
rt_grid.fit(cp_train, cp_train_Y)
print(f'Best hyper-parameters: {rt_grid.best_params_} and inner evaluation: {rt_grid.best_score_}')
```

Hyper-parameter tuning

| Algorithm | Best parameter | Best score |
|---|---|---|
| KNN | Neighbors: 14 | **-2.394.762,7526418786** |
| SVM | C: 159,61869340759762 | -3.290.533,430136986 |
| | class_weight: balanced | |
| | gamma: 0.000135929571 | |
| | kernel: rbf | |
| Regression Tree | min_samples_split: 8 | -2.561.567,0108509036 |
| | min_samples_lea: 8 | |
| | max_leaf_nodes: 14 | |
| | max_depth: 6 | |

We can observed in the following table the best approach out of the six methods:

| | ALGORITHM | MEAN ABSOLUTE ERROR |
|---|---|---|
| WITHOUT HYPER-PARAMETER TUNING | KNN | 2.588.455,571506849 |
| | SVM | 6.920.271,25479452 |
| | Regression Tree | 3.078.653,835616438 |
| WITH HYPER-PARAMETER TUNING | KNN | **-2.394.762,7526418786** |
| | SVM | -3.290.533,430136986 |
| | Regression Tree | -2.561.567,0108509036 |

**MODEL EVALUATION**

According to the table above, the best model is  KNN with hyper-parameters tuning (neighbors = 14). We will use it to evaluate the test set, which has been scaled in order to be useful in the knn model.

```
In [2]: prediction = knn_grid.predict(scaled_test_x)
   ...: print("MAE on Test Set:"
   ...:       , metrics.mean_absolute_error(prediction, cp_test_Y))
MAE on Test Set: 2271745.778600663
```

The MAE over the test set is **2,271,745.778600663**. The model predicts better this information in comparison with the validation_train data set.

**FINAL MODEL**

Use all available data to train the final model. KNN with 14 neighbors.

```
In [4]: data_x = pd.concat([cp_train, cp_test])
   ...: data_y = pd.concat([cp_train_Y, cp_test_Y])
   ...: final_knn = neighbors.KNeighborsRegressor(n_neighbors=14)
   ...: data_x_scaled = scaler.transform(data_x)
   ...: final_knn.fit(data_x_scaled, data_y)
   ...: print("MAE on Test Set:",
   ...:       metrics.mean_absolute_error(data_y, final_knn.predict(data_x_scaled)))
MAE on Test Set: 2168074.196362214
```

The MAE over the whole available data is **2,168,074.196362214**, the most accurate model of the ones tested in this practice.