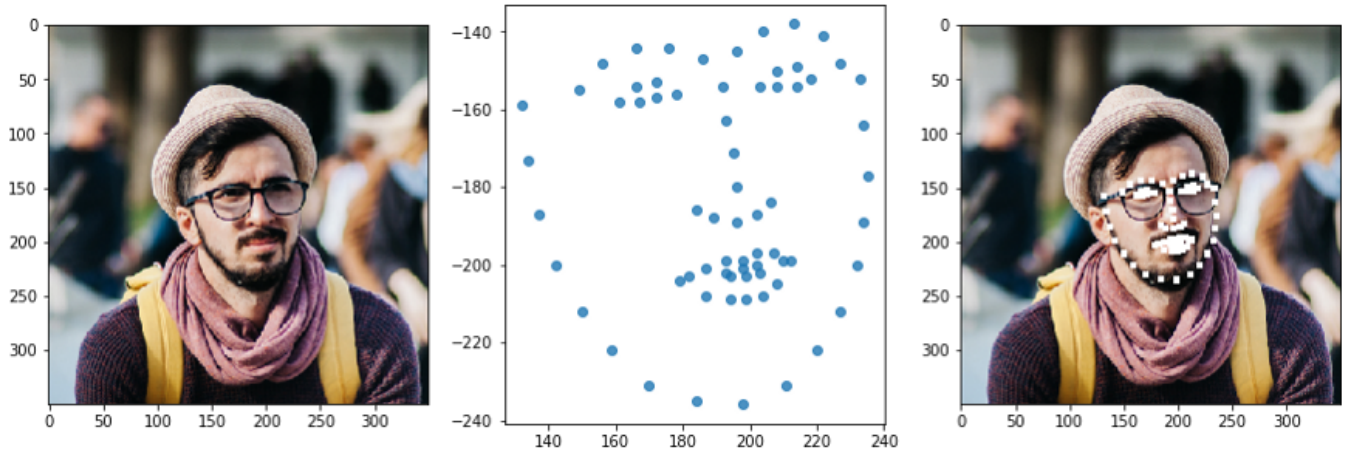
MLxtend: Feature Selection Tutorial

put together by KVS Setty (kvssetty.com)on 24th Aug 2020 in Data Science (https://kvssetty.com/category/data-science/)

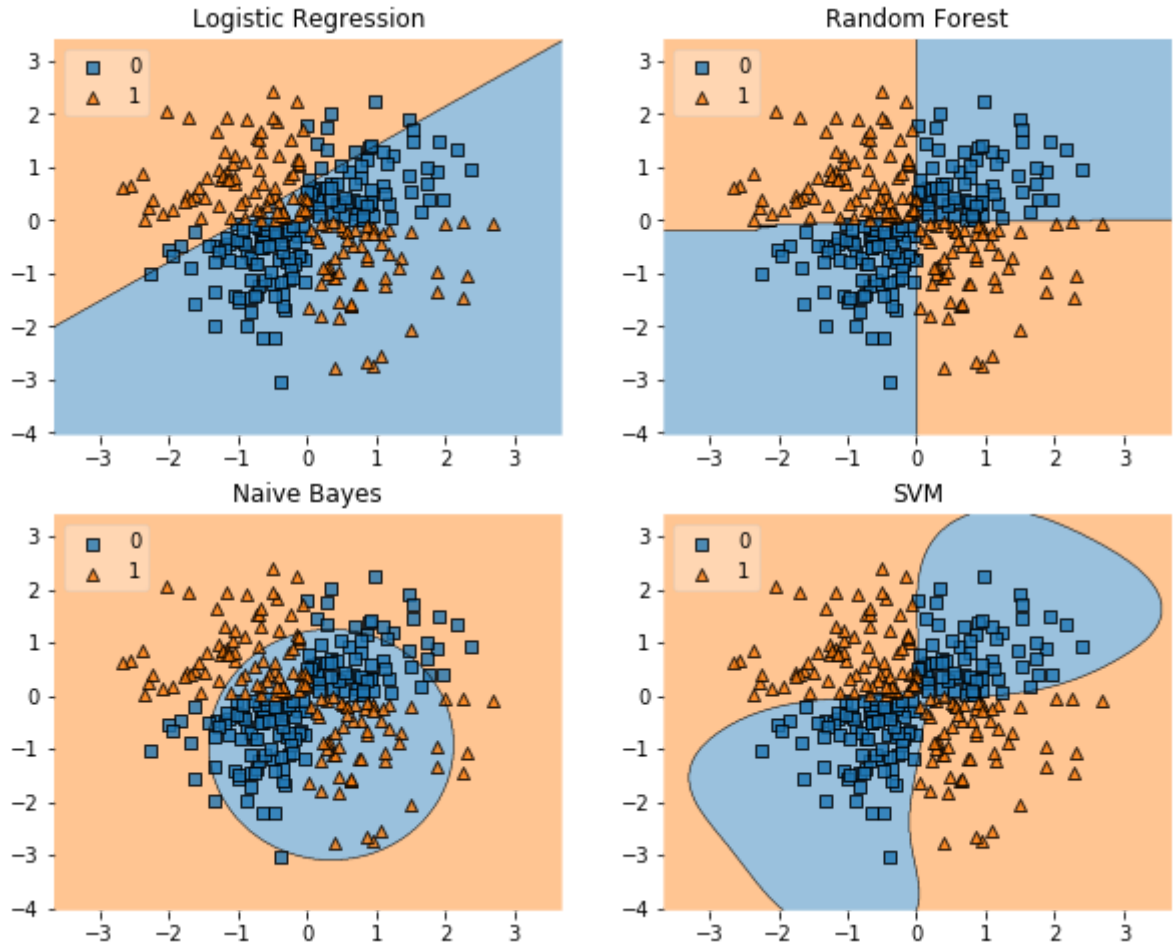
What is MLxtend ?

MLxtend is a library that should accompany any data science project. Considered as an extension of the Sci-kit learn library, MLxtend has useful automation of common data science tasks:

- Completely automated feature extraction and selection.
- An extension on Sci-kit learn's existing data transformers, like mean centering and transaction encoders.
- A vast array of evaluation metrics: a few include bias-variance decomposition (measure how bias and variance your model contains), lift tests, McNemar's test, F-test, and many more.
- Helpful model visualizations, including feature boundaries, learning curves, PCA correlation circles, and enrichment plots.
- Many built-in datasets that are not included in Sci-kit Learn.
- Helpful preprocessing functions for images and text, like a name generalizer that can identify and convert text with different naming systems ("Deer, John", "J. Deer", "J. D.", and "John Deer" are the same).

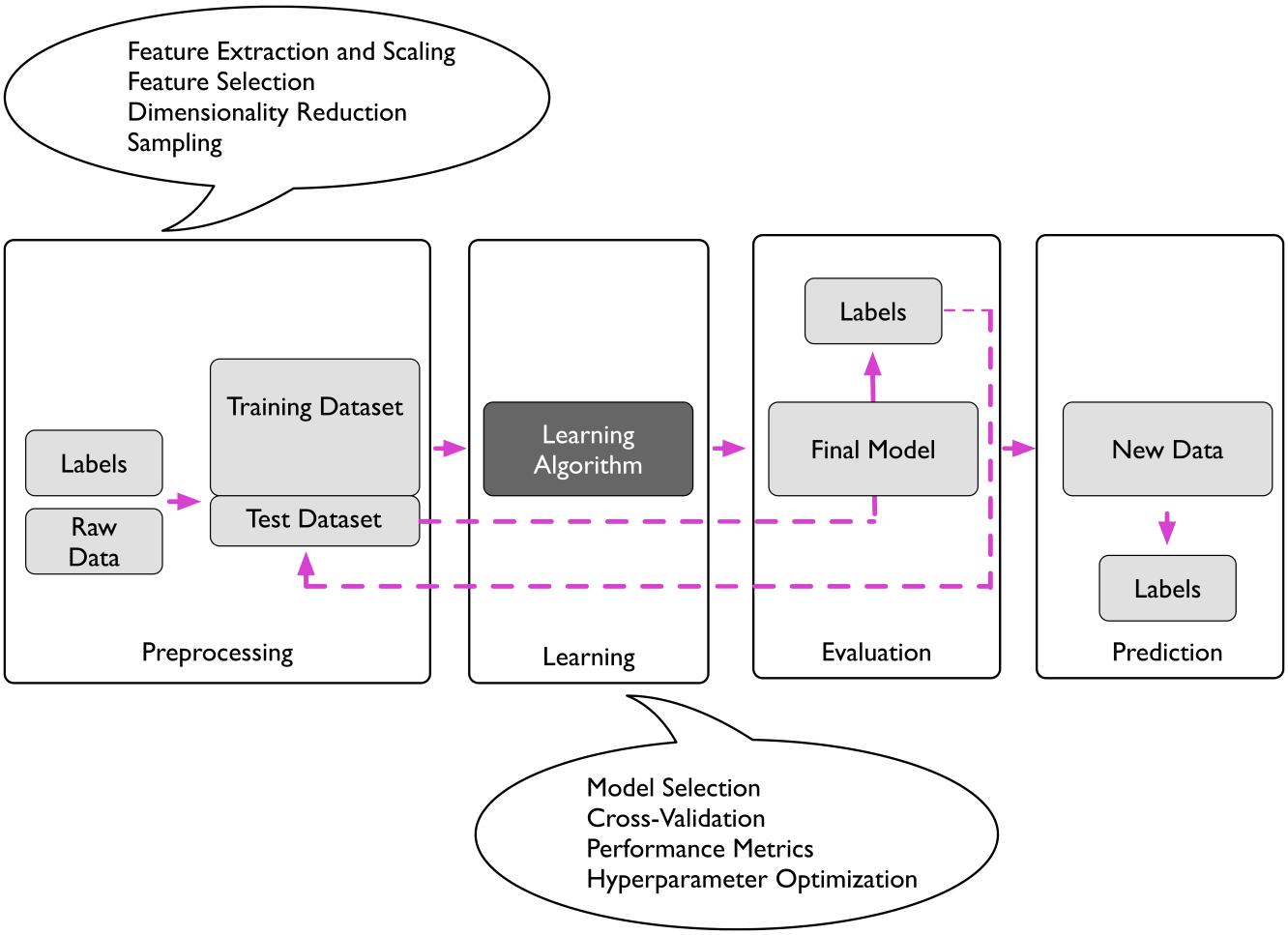


Or, consider its decision boundary drawing capabilities:



For more info on its features see the **Documentation** (http://rasbt.github.io/mlxtend/)

In this article, I present MLxtend (machine learning extensions), a Python library of useful tools for the day-to-day data science tasks. To showcase its strength I use the library to select the most important features of a dataset before feeding data into learning algorithm.Feature selection is an preprocessing step, see the fig below.


Feature Extraction and Scaling
Feature Selection
Dimensionality Reduction
Sampling
Labels
Raw Data
Training Dataset
Test Dataset
Preprocessing
Learning Algorithm
Learning
Labels
Final Model
Evaluation
New Data
Labels
Prediction
Model Selection
Cross-Validation
Performance Metrics
Hyperparameter Optimization

Tutorial Overview

This tutorial is divided into ten parts; they are:

- How to install it?
- Curse of dimensionality
- Feature Selection
- Exhaustive search
- Forward feature selection
- Backward feature selection
- Stochastic feature selection
- Python Implementation
- Summary
- Further Reading

How to install it?

Just run the following command if you have conda installed in your PC:

```
conda install -c conda-forge mlxtend
```

Or using pip:

```
pip install mlxtend
```

MLxtend is a useful package for diverse data science-related tasks. It contains some useful wrapper methods such as:

- **SequentialFeatureSelector** (supporting both Forward and Backward feature selection)
- **ExhaustiveFeatureSelector**

Curse of dimensionality

Being in the know that adding more features is not always helpful. This is due to:

Data is sparse in high dimensions Impractical to use all the measured data directly Some features may be detrimental to pattern recognition Some features are essentially noise

💀 Curse of dimensionality

Being in the know that adding more features is not always helpful. This is due to:

- Data is sparse in high dimensions
- Impractical to use all the measured data directly
- Some features may be detrimental to pattern recognition
- Some features are essentially noise

> But some may not be relevant to the outcome. Moreover, many of the original predictors also may not contain predictive information. For a number of models, predictive performance is degraded as the number of uninformative predictors increases. Therefore, there is a genuine need to appropriately select predictors for modeling.

--page 227, Chapter 10: Feature Engineering and selection by Max Kuhun (https://www.amazon.in/Feature-Engineering-Selection-Practical-Predictive/dp/1138079227/ref=sr_1_1?dchild=1&keywords=feature+engineering+for+machine+learning+by+max+kuhn&qid=1598246314&s=books&s1)

🧠 Feature Selection

Feature Selection is the process of selecting a subset of the extracted features. This is helpful because:

- Reduces dimensionality
- Discards uninformative features
- Discards deceptive features (Deceptive features appear to aid learning on the training set, but impair generalisation)
- Speeds training/testing

> The working premise here is that it is generally better to have fewer predictors in a model.[...], the goal of feature selection will be re-framed to
>
> **Reduce the number of predictors as far as possible without compromising predictive performance.**

--page 228, Chapter 10: Feature Engineering and selection by Max Kuhun (https://www.amazon.in/Feature-Engineering-Selection-Practical-Predictive/dp/1138079227/ref=sr_1_1?dchild=1&keywords=feature+engineering+for+machine+learning+by+max+kuhn&qid=1598246314&s=books&s1)

Types of Feature Selection Methodologies

Feature selection mythologies fall into three general classes:

- **Intrinsic (or implicit) methods.**
- **Filter methods.**
- **Wrapper methods.**

Intrinsic methods have feature selection naturally incorporated with the modeling process. Whereas filter and wrapper methods work to marry feature selection approaches with modeling techniques.The important of the three classes for reducing features are wrapper methods. And this tutorial uses and explores wrapper methods,so some explanation is well worth.

Wrapper methods use iterative search procedures that repeatedly supply predictor subsets to the model and then use the resulting model performance estimate to guide the selection of the next subset to evaluate. If successful, a wrapper method will iterate to a smaller set of predictors that has better predictive performance than the original predictor set. Wrapper methods can take either a greedy or non-greedy approach to feature selection. A greedy search is one that chooses the search path based on the direction that seems best at the time in order to achieve the best immediate benefit. While this can be an effective strategy, it may show immediate benefits in predictive performance that stall out at a locally best setting. A non-greedy search method would re-evaluate previous feature combinations and would have the ability to move in a direction that is initially unfavorable if it appears to have a potential benefit after the current step. This allows the non-greedy approach to escape being trapped in a local optima.

An example of a greedy wrapper method is **backwards selection (otherwise known as recursive feature elimination or RFE)**. Here, the predictors are initially ranked by some measure of importance. An initial model is created using the complete predictor set. The next model is based on a smaller set of predictors where the least important have been removed. This process continues down a prescribed path (based on the ranking generated by the importances) until a very small number of predictors are in the model. Performance estimates are used to determine when too many features have been removed; hopefully a smaller subset of predictors can result in an improvement. Notice that the RFE procedure is greedy in that it considers the variable ranking as the search direction. It does not re-evaluate the search path at any point or consider subsets of mixed levels of importance. This approach to feature selection will likely fail if there are important interactions between predictors where only one of the predictors is significant in the presence of the other(s).

Examples of non-greedy wrapper methods (also called Stochastic feature selection) are **genetic algorithms (GA)** and **simulated annealing (SA).** The SA method is non-greedy since it incorporate randomness into the feature selection process. The random component of the process helps SA to find new search spaces that often lead to more optimal results.

Wrappers have the potential advantage of searching a wider variety of predictor subsets than simple filters or models with built-in intrinsic(implicit) feature selection. They have the most potential to find the globally best predictor subset (if it exists). The primary drawback is the computational time required for these methods to find the optimal or near optimal subset. The additional time can be excessive to the extent of being counter-productive. The computational time problem can be further exacerbated by the type of model with which it is coupled. For example, the models that are in most need of feature selection (e.g., SVMs and neural networks) can be very computationally taxing themselves. Another disadvantage of wrappers is that they have the most potential to overfit the predictors to the training data and require external validation

In general, there are three wrapper approach which we will analyze in more details shortly are:

- Exhaustive search generally too expensive
- Forward/backward greedy search algorithms
- Stochastic search (Simulated Annealing and Genetic Algorithms)

Exhaustive search

The goal is:

Given M input features, select a subset of the d most useful. Try each combination of d features and assess which is most effective. Number of combinations:

$$M!/(M-d)d!$$

Allowing subsets of size d = 1, . . . , M gives $2^M - 1$ combinations. Prohibitively expensive for M >= 20 (2^(20) ≈ 1,000,000). Since it is potentially too expensive. Forward and backward(RFE) are usually the preferred option.

Forward Feature Selection (FFS)

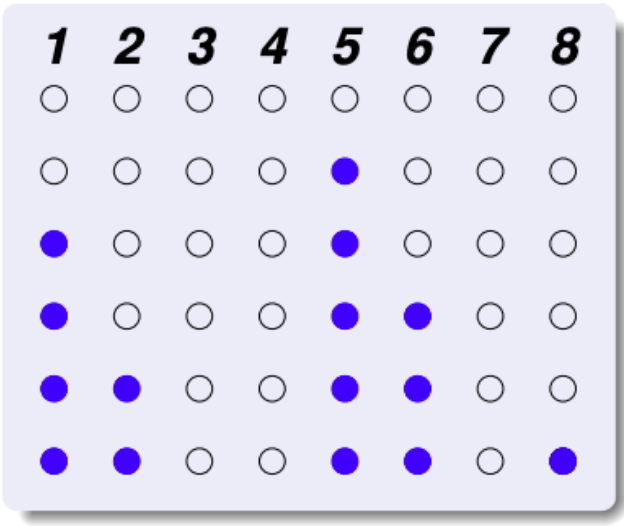
Forward feature selection

Greedy search

Initialise selected set $S = \emptyset$

Initialise unselected set $U = \{1, \ldots, M\}$

- Evaluate performance with $S \cup u_i$ for each $u_i \in U$
- $S := S \cup u_m$ and $U := U \setminus u_m$ where u_m gives maximum improvement in performance
- Stop when no significant improvement in classification or d features.



Does not guarantee that another untried feature set is not better

The forward selection involves the below steps:

- Train the model with a single feature the one which gives the better result based on the evaluation metric.
- Select a second feature which in combination with the first gives the best performance.
- Continue the above steps
- Stop when no significant improvement is observed or the limit of d features is observed.

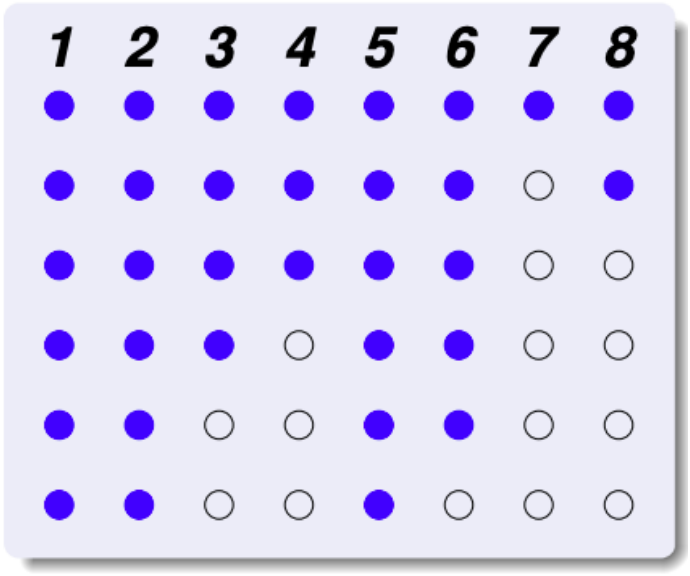
Backward Feature Selection (a.k.a Recurssive Feature Elimination ,RFE)

Backward feature selection

Initialise selected set $S = \{1, \ldots, M\}$
Initialise unselected set $U = \emptyset$

Repeat

- Evaluate performance with $S \setminus x_i$ for each $x_i \in S$
- $S := S \setminus x_m$ and $U := U \cup x_m$ where x_m gives maximum improvement/minimum decrease in performance
- Stop when no significant improvement or d features



Does not guarantee that another untried feature set is not better

The backward selection involves the below steps:

- Train the model using all features
- Discard the one which gives the least decrease in the performance
- Continue the above steps
- Stop when significant decrease of the performance is observed or the limit of d features is observed.

Both techniques are fast but does not guarantee that another untried feature set is not better (thus it is greedy search). It is only guarantee that eliminate features whose information content is subsumed by other features.

Stochastic feature selection (Simulated Annealing and Genetic Algorithms)

Feature selection is a combinatorial optimization problem:

- Simulated annealing(SA) or genetic algorithms(GA) to locate global maximum
- Potentially very good results
- Potentially very expensive

Python Implementation in MLxtend

Lets see some examples of above algorithms in action implemented mlxtend:

Exhaustive Feature Selector

Implementation of an exhaustive feature selector for sampling and evaluating all possible feature combinations in a specified range.

```
from mlxtend.feature_selection import ExhaustiveFeatureSelector
```

Overview This exhaustive feature selection algorithm is a wrapper approach for brute-force evaluation of feature subsets; the best subset is selected by optimizing a specified performance metric given an arbitrary regressor or classifier. For instance, if the classifier is a logistic regression and the dataset consists of 4 features, the alogorithm will evaluate all 15 feature combinations (if `min_features=1` and `max_features=4`)

- {0}
- {1}
- {2}
- {3}
- {0, 1}
- {0, 2}
- {0, 3}
- {1, 2}
- {1, 3}
- {2, 3}
- {0, 1, 2}
- {0, 1, 3}
- {0, 2, 3}
- {1, 2, 3}
- {0, 1, 2, 3}

and select the one that results in the best performance (e.g., classification accuracy) of the logistic regression classifier.

Example 1 - A simple Iris example

Initializing a simple classifier from scikit-learn:

In [1]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS

iris = load_iris()
X = iris.data
y = iris.target

knn = KNeighborsClassifier(n_neighbors=3)

efs1 = EFS(knn,
           min_features=1,
           max_features=4,
           scoring='accuracy',
           print_progress=True,
           cv=5)

efs1 = efs1.fit(X, y)

print('Best accuracy score: %.2f' % efs1.best_score_)
print('Best subset (indices):', efs1.best_idx_)
print('Best subset (corresponding names):', efs1.best_feature_names_)
```

```
Features: 15/15

Best accuracy score: 0.97
Best subset (indices): (0, 2, 3)
Best subset (corresponding names): ('0', '2', '3')
```

Note that in the example above, the 'best_feature*names*' are simply a string equivalent of the feature indices. However, we can provide custom feature names to the fit function for this mapping:

In [2]:

```
feature_names = ('sepal length', 'sepal width', 'petal length', 'petal width')
efs1 = efs1.fit(X, y, custom_feature_names=feature_names)
print('Best subset (corresponding names):', efs1.best_feature_names_)
```

```
Features: 15/15

Best subset (corresponding names): ('sepal length', 'petal length', 'petal
width')
```

Example 02 - Working with pandas DataFrames

Optionally, we can also use pandas DataFrames and pandas Series as input to the fit function. In this case, the column names of the pandas DataFrame will be used as feature names. However, note that if custom_feature_names are provided in the fit function, these `custom_feature_names` take precedence over the DataFrame column-based feature names.

In [3]:

```
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris

iris = load_iris()
col_names = ('sepal length', 'sepal width',
             'petal length', 'petal width')
X_df = pd.DataFrame(iris.data, columns=col_names)
y_series = pd.Series(iris.target)
knn = KNeighborsClassifier(n_neighbors=4)
```

In [4]:

```
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS

knn = KNeighborsClassifier(n_neighbors=3)

efs1 = EFS(knn,
           min_features=1,
           max_features=4,
           scoring='accuracy',
           print_progress=True,
           cv=5)

efs1 = efs1.fit(X_df, y_series)

print('Best accuracy score: %.2f' % efs1.best_score_)
print('Best subset (indices):', efs1.best_idx_)
print('Best subset (corresponding names):', efs1.best_feature_names_)
```

```
Features: 15/15

Best accuracy score: 0.97
Best subset (indices): (0, 2, 3)
Best subset (corresponding names): ('sepal length', 'petal length', 'petal
width')
```

Sequential Feature Selector

Implementation of sequential feature algorithms (SFAs) -- greedy search algorithms -- that have been developed as a suboptimal solution to the computationally often not feasible exhaustive search.

```
from mlxtend.feature_selection import SequentialFeatureSelector
```

Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce an initial d-dimensional feature space to a k-dimensional feature subspace where k < d. The motivation behind feature selection algorithms is to automatically select a subset of features that is most relevant to the problem. The goal of feature selection is two-fold: We want to improve the computational efficiency and reduce the generalization error of the model by removing irrelevant features or noise. A wrapper approach such as sequential feature selection is especially useful if intrinsic (implicit) feature selection -- for example, a regularization penalty like LASSO -- is not applicable.

In a nutshell, SFAs remove or add one feature at the time based on the classifier performance until a feature subset of the desired size k is reached. There are 4 different flavors of SFAs available in mlxtend via the `SequentialFeatureSelector`:

- Sequential Forward Selection (SFS)
- Sequential Backward Selection (SBS)
- Sequential Forward Floating Selection (SFFS)
- Sequential Backward Floating Selection (SBFS)

The **floating** variants, SFFS and SBFS, can be considered as extensions to the simpler SFS and SBS algorithms. The floating algorithms have an additional exclusion or inclusion step to remove features once they were included (or excluded), so that a larger number of feature subset combinations can be sampled. It is important to emphasize that this step is conditional and only occurs if the resulting feature subset is assessed as "better" by the criterion function after removal (or addition) of a particular feature. Furthermore, I added an optional check to skip the conditional exclusion steps if the algorithm gets stuck in cycles.
Important Note: How is this different from Recursive Feature Elimination (RFE) -- e.g., as implemented in `sklearn.feature_selection.RFE`? RFE is computationally less complex using the feature weight coefficients (e.g., linear models) or feature importance (tree-based algorithms) to eliminate features recursively, whereas SFSs eliminate (or add) features based on a user-defined classifier/regression performance metric.

Example 1 - A simple Sequential Forward Selection example

Initializing a simple classifier from scikit-learn:

In [5]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target
knn = KNeighborsClassifier(n_neighbors=4)
```

We start by selection the "best" 3 features from the Iris dataset via Sequential Forward Selection (SFS). Here, we set `forward=True` and `floating=False`. By choosing `cv=0`, we don't perform any cross-validation, therefore, the performance (here: `accuracy`) is computed entirely on the training set.

In [6]:

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

sfs1 = SFS(knn,
           k_features=3,
           forward=True,
           floating=False,
           verbose=2,
           scoring='accuracy',
           cv=0)

sfs1 = sfs1.fit(X, y)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent wo
rkers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:
0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s finished

[2020-08-24 14:08:28] Features: 1/3 -- score: 0.96[Parallel(n_jobs=1)]: Us
ing backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:
0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s finished

[2020-08-24 14:08:28] Features: 2/3 -- score: 0.9733333333333334[Parallel
(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:
0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s finished

[2020-08-24 14:08:28] Features: 3/3 -- score: 0.9733333333333334
```

Via the `subsets_` attribute, we can take a look at the selected feature indices at each step:

In [7]:

```
sfs1.subsets_
```

Out[7]:

```
{1: {'feature_idx': (3,),
  'cv_scores': array([0.96]),
  'avg_score': 0.96,
  'feature_names': ('3',)},
 2: {'feature_idx': (2, 3),
  'cv_scores': array([0.97333333]),
  'avg_score': 0.9733333333333334,
  'feature_names': ('2', '3')},
 3: {'feature_idx': (1, 2, 3),
  'cv_scores': array([0.97333333]),
  'avg_score': 0.9733333333333334,
  'feature_names': ('1', '2', '3')}}
```

Note that the 'feature_names' entry is simply a string representation of the 'feature_idx' in this case. Optionally, we can provide custom feature names via the `fit` method's `custom_feature_names` parameter:

In [8]:

```
feature_names = ('sepal length', 'sepal width', 'petal length', 'petal width')
sfs1 = sfs1.fit(X, y, custom_feature_names=feature_names)
sfs1.subsets_
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent wo
rkers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:
0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s finished

[2020-08-24 14:08:28] Features: 1/3 -- score: 0.96[Parallel(n_jobs=1)]: Us
ing backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:
0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s finished

[2020-08-24 14:08:28] Features: 2/3 -- score: 0.9733333333333334[Parallel
(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:
0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s finished

[2020-08-24 14:08:28] Features: 3/3 -- score: 0.9733333333333334
```

Out[8]:

```
{1: {'feature_idx': (3,),
  'cv_scores': array([0.96]),
  'avg_score': 0.96,
  'feature_names': ('petal width',)},
 2: {'feature_idx': (2, 3),
  'cv_scores': array([0.97333333]),
  'avg_score': 0.9733333333333334,
  'feature_names': ('petal length', 'petal width')},
 3: {'feature_idx': (1, 2, 3),
  'cv_scores': array([0.97333333]),
  'avg_score': 0.9733333333333334,
  'feature_names': ('sepal width', 'petal length', 'petal width')}}
```

Furthermore, we can access the indices of the 3 best features directly via the `k_feature_idx_` attribute:

In [9]:

```
sfs1.k_feature_idx_
```

Out[9]:

```
(1, 2, 3)
```

And similarly, to obtain the names of these features, given that we provided an argument to the `custom_feature_names` parameter, we can refer to the `sfs1.k_feature_names_` attribute:

In [10]:

```
sfs1.k_feature_names_
```

Out[10]:

```
('sepal width', 'petal length', 'petal width')
```

Finally, the prediction score for these 3 features can be accesses via `k_score_` :

In [11]:

```
sfs1.k_score_
```

Out[11]:

```
0.9733333333333334
```

Summary

This brings us to the end of this article. Hope you become aware of the MLxtend Python library and how it can be used for feature selection and it has ton other modules for every day data science use and highly recommended get familiar with it.

In this tutorial, you discovered and specifically learned :

- what is feature selection?
- why we need it and what problems it solves?
- the various methods available.
- Imlementation in Python MLxtend Library.
- How to use them with some examples.

Further Reading

For book lovers:

Python for Data Analysis (https://www.amazon.in/Python-Data-Analysis-Wes-Mckinney/dp/1491957662/ref=sr_1_fkmr0_1?dchild=1&keywords=Python+for+Data+Analysis%2C+2e%3A+Data+Wrangling+with+Pandas%2C+Numpy%2C&qid=1598246314&s=books&sr=1-fkmr0) by Wes McKinney, best known for creating the Pandas project.

Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (https://www.amazon.in/Hands-Machine-Learning-Scikit-Learn-Tensor/dp/9352139054/ref=pd_lpo_14_img_0/257-8149962-4834767?_encoding=UTF8&pd_rd_i=9352139054&pd_rd_r=c78d3dad-fa65-4819-8e3b-725c920337d3&pd_rd_w=z5O01&pd_rd_wg=6jtrj&pf_rd_p=5a903e39-3cff-40f0-9a69-33552e242181&pf_rd_r=0F0HE57N6RVF44X0B2X0&psc=1&refRID=0F0HE57N6RVF44X0B2X0) by Aurelien Geron, currently ranking first in the best sellers Books in AI & Machine Learning on Amazon.

Feature Engineering and selection by Max Kuhun (https://www.amazon.in/Feature-Engineering-Selection-Practical-Predictive/dp/1138079227/ref=sr_1_1?dchild=1&keywords=feature+engineering+for+machine+learning+by+max+kuhn&qid=1598246314&s=books&sr=1-1) by Max Kuhn and Kjell Johnson

Python Machine Learning (https://www.amazon.in/Python-Machine-Learning-scikit-learn-TensorFlow/dp/1789955750/ref=sr_1_4?crid=3G4DEQC0XILTL&dchild=1&keywords=python+machine+learning+sebastian+raschka&qid=1598257497&sr=8-4): Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2.0, 3rd Edition by Sebastian Raschka

Do you have any questions?

Ask your questions in the comments/reponses column and I will do my best to answer.

Get the complete source code from my git page: https://github.com/KVSSetty/MLxtend-Tutorials (https://github.com/KVSSetty/MLxtend-Tutorials)

or sign-up for my week-end online classes : https://www.mlanddlguru.com/b/signup (https://www.mlanddlguru.com/b/signup)