Sebastian Raschka
last updated: **03/29/2014**
Link to this IPython Notebook on GitHub (https://github.com/rasbt/algorithms_in_ipython_notebooks)

Executed in Python 3.4.0

Sections

Sequential Selection Algorithms in Python

1. Introduction

[back to top]

One of the biggest challenges of designing a good classifier for solving a ***Statistical Pattern Classification*** problem is to estimate the underlying parameters to fit th model - given that the forms of the underlying probability distributions are known. The larger the number of parameters becomes, the more difficult it naturally is to estimat those parameters accurately (***Curse of Dimensionality***) from a limited number of training samples.

In order to avoid the ***Curse of Dimensionality***, pattern classification is often accompanied by ***Dimensionality Reduction***, which also has the nice side-effect of increasin the computational performance. Common techniques are projection-based, such as ***Principal Component Analysis (PCA)***, ***Linear Dimension Analysis (LDA)***, an ***Multivariate Dimension Analysis (MDA)***.
An alternative to the projection-based approach is the so-called ***Feature Selection***, and in this article, we will take a look at some of the established algorithms to tack this combinatorial search problem. Note that those algorithms are considered as "subpoptimal" in contrast to an ***exhaustive search***, which is often computationally nc feasible, though.

Therefore, the goal of the presented ***sequential selection algorithms*** is to reduce the feature space $D = \{x_1, x_2, x_n\}$ to a subset of features $D - n$ in order to improve c optimize the **computational performance** of the classifier and to avoid the so-called ***Curse of Dimensionality***.
The goal is to select a "sufficiently reduced" subset from the feature space D "without significantly reducing" the performance of the classifier. In the process of choosing a "optimal" feature subset of size k, a so-called ***Criterion Function***, which typically, simply, and intuitively assesses the ***recognition rate*** of the classifier.

F. Ferri, P. Pudil, M. Hatef, and J. Kittler investigated the performance of different ***Sequential Selection Algorithms*** for ***Feature Selection*** on different scales and reporte their results in a nice research article: "*Comparative Study of Techniques for Large Scale Feature Selectio (http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=02CB16CB1C28EA6CB57E212861CFB180?doi=10.1.1.24.4369&rep=rep1&type=pdf)*," *Pattern Recognition Practice IV, E. Gelsema and L. Kanal, eds., pp. 403-413. Elsevier Science B.V., 1994.*
Choosing an "appropriate" algorithm really depends on the problem - the size and desired recognition rate and computational performance. Thus, I want to encourage yo to take (at least) a brief look at their paper and the results they obtained from experimenting with different problems feature space dimensions.

Defining a criterion function (for testing)

[back to top]

In order to evaluate the performance of our selected ***feature subset*** (typically the recognition rate of the classifier), we need to define a ***criterion function***.

For the sake of simplicity, and in order to get an intuitive idea if our algorithm returns an "appropriate" result, let us define a very simple criterion function here.
The criterion function defined below simply returns the sum of numerical values in a list.

In [1]:
```
def simple_crit_func(feat_sub):
    """ Returns sum of numerical values of an input list. """
    return sum(feat_sub)

# Example:
simple_crit_func([1,2,4])
```

Out[1]: 7

2. Sequential Forward Selection (SFS)

[back to top]

The ***Sequential Fortward Selection (SFS)*** is one of the simplest and probably fastest *Feature Selection* algorithms.
Let's summarize its mechanics in words:
SFS starts with an empty feature subset and sequentially adds features from the whole input feature space to this subset until the subset reaches a desired (user-specifie size. For every iteration (= inclusion of a new feature), the whole feature subset is evaluated (expect for the features that are already included in the new subset). Th evaluation is done by the so-called ***criterion function*** which assesses the feature that leads to the maximum performance improvement of the feature subset if it included.
Note that included features are never removed, which is one of the biggest downsides of this algorithm.

Input: the set of all features, *Y = {y_1, y_2, ..., y_d}*

- The ***SFS*** algorithm takes the whole feature set as input, if our feature space consists of, e.g. 10, if our feature space consists of 10 dimensions (***d = 10***).

Output: a subset of features, *X_k = {x_j | j = 1, 2, ..., k; x_j ∈ Y}*, where *k = (0, 1, 2, ..., d*)

- The returned output of the algorithm is a subset of the feature space of a specified size. E.g., a subset of 5 features from a 10-dimensional feature space (***k = 5, d = 10***).

Initialization: *X_0 = "null set", k = 0*

- We initialize the algorithm with an empty set ("null set") so that the ***k = 0*** (where ***k*** is the size of the subset)

Step 1 (Inclusion):

x^+ arg max J(x_k + x), where x ∈ Y - X_k
X_k+1 = X_k + x^+
k = k + 1
Go to Step 1

- We go through the ***feature space*** and look for the feature ***x^+*** which maximizes our criterion if we add it to the ***feature subset*** (where ***J()*** is the criterion function). We repeat this process until we reach the ***Termination*** criterion.

Termination: stop when *k* equals the number of desired features

- We add features to the new feature subset ***X_k*** until we reach the number of specified features for our final subset. E.g., if our desired number of features is 5 and we start with the "null set" (*Initialization*), we would add features to the subset until it contains 5 features.

SFS Code

[back to top]

In [2]:
```
# Sebastian Raschka
```

```
# last updated: 03/29/2014
# Sequential Forward Selection (SBS)


def seq_forw_select(features, max_k, criterion_func, print_steps=False):
    """
    Implementation of a Sequential Forward Selection algorithm.

    Keyword Arguments:
        features (list): The feature space as a list of features.
        max_k: Termination criterion; the size of the returned feature subset.
        criterion_func (function): Function that is used to evaluate the
            performance of the feature subset.
        print_steps (bool): Prints the algorithm procedure if True.

    Returns the selected feature subset, a list of features of length max_k.

    """

    # Initialization
    feat_sub = []
    k = 0
    d = len(features)
    if max_k > d:
        max_k = d

    while True:

        # Inclusion step
        if print_steps:
            print('\nInclusion from feature space', features)
        crit_func_max = criterion_func(feat_sub + [features[0]])
        best_feat = features[0]
        for x in features[1:]:
            crit_func_eval = criterion_func(feat_sub + [x])
            if crit_func_eval > crit_func_max:
                crit_func_max = crit_func_eval
                best_feat = x
        feat_sub.append(best_feat)
        if print_steps:
            print('include: {} -> feature subset: {}'.format(best_feat, feat_sub))
        features.remove(best_feat)

        # Termination condition
        k = len(feat_sub)
        if k == max_k:
            break

    return feat_sub
```

Example SFS:

[back to top]

In this example, we take a look at the individual steps of the ***SFS*** algorithmn to select a ***feature subset*** consisting of 3 features out of a ***feature space*** of size 10.
The input feature space consists of 10 integers: 6, 3, 1, 6, 8, 2, 3, 7, 9, 1,
and our criterion is to find a subset of size 3 in this ***feature space*** that maximizes the integer sum in this ***feature subset***.

In [3]:
```
def example_seq_forw_select():
    ex_features = [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
    res_forw = seq_forw_select(features=ex_features, max_k=3,\
                               criterion_func=simple_crit_func, print_steps=True)
    return res_forw

# Run example
res_forw = example_seq_forw_select()
print('\nRESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] ->', res_forw)
```

```
Inclusion from feature space [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
include: 9 -> feature subset: [9]

Inclusion from feature space [6, 3, 1, 6, 8, 2, 3, 7, 1]
include: 8 -> feature subset: [9, 8]

Inclusion from feature space [6, 3, 1, 6, 2, 3, 7, 1]
include: 7 -> feature subset: [9, 8, 7]

RESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] -> [9, 8, 7]
```

Result:

The returned result is definitely what we would expect: the 3 highest values (note that we defined 3 as the number of desired features for our subset) in the input feature lis since our ***criterion*** is to select the numerical values (= features) that yield the maximum mathematical sum in the feature subset.

---

3. Sequential Backward Selection (SBS)

[back to top]

The ***Sequential Backward Selection (SBS)*** algorithm is very similar to the ***SFS***, which we have just seen in the section above. The only difference is that we start with th complete feature set instead of the "null set" and remove features sequentially until we reach the number of desired features *k*.
Note that features are never added back once they were removed, which (similar to ***SFS***) is one of the biggest downsides of this algorithm.

Input: the set of all features, $Y = \{y_1, y_2, ..., y_d\}$

- The ***SBS*** algorithm takes the whole feature set as input, if our feature space consists of, e.g. 10, if our feature space consists of 10 dimensions (***d = 10***).

Output: a subset of features, $X_k = \{x_j \mid j = 1, 2, ..., k; x_j \in Y\}$, where $k = (0, 1, 2, ..., d)$

- The returned output of the algorithm is a subset of the feature space of a specified size. E.g., a subset of 5 features from a 10-dimensional feature space (***k = 5, d = 10***).

Initialization: $X_0 = Y$, $k = d$

- We initialize the algorithm with the given feature set so that the ***k = d*** (where ***k*** has the size of the feature set ***d***)

Step 1 (Exclusion):

x^- = arg max J(x_k - x), where x ∈ X_k
X_k-1 = X_k - x^-
k = k - 1
Go to Step 1

- We go through the ***feature subset*** and look for the feature ***x^-*** which minimizes our criterion if we remove it from the ***feature subset*** (where ***J()*** is the criterion function). We repeat this process until we reach the ***Termination*** criterion.

Termination: stop when *k* equals the number of desired features

- We remove features from the feature subset ***X_k*** until we reach the number of specified features for our final subset. E.g., if our desired number of features is 5 and we start with the entire feature space (*Initialization*), we would remove features from the subset until it contains 5 features.

SBS Code

[back to top]

In [4]:
```
# Sebastian Raschka
# last updated: 03/29/2014
# Sequential Backward Selection (SBS)

from copy import deepcopy

def seq_backw_select(features, max_k, criterion_func, print_steps=False):
    """
    Implementation of a Sequential Backward Selection algorithm.

    Keyword Arguments:
        features (list): The feature space as a list of features.
        max_k: Termination criterion; the size of the returned feature subset.
        criterion_func (function): Function that is used to evaluate the
            performance of the feature subset.
        print_steps (bool): Prints the algorithm procedure if True.

    Returns the selected feature subset, a list of features of length max_k.
```

```
    """
    # Initialization
    feat_sub = deepcopy(features)
    k = len(feat_sub)
    i = 0

    while True:

        # Exclusion step
        if print_steps:
            print('\nExclusion from feature subset', feat_sub)
        worst_feat = len(feat_sub)-1
        worst_feat_val = feat_sub[worst_feat]
        crit_func_max = criterion_func(feat_sub[:-1])

        for i in reversed(range(0,len(feat_sub)-1)):
            crit_func_eval = criterion_func(feat_sub[:i] + feat_sub[i+1:])
            if crit_func_eval > crit_func_max:
                worst_feat, crit_func_max = i, crit_func_eval
                worst_feat_val = feat_sub[worst_feat]
        del feat_sub[worst_feat]
        if print_steps:
            print('exclude: {} -> feature subset: {}'.format(worst_feat_val, feat_sub))

        # Termination condition
        k = len(feat_sub)
        if k == max_k:
            break

    return feat_sub
```

Example SBS:

[back to top]

Like we did for the ***SFS example*** above, we take a look at the individual steps of the ***SBS*** algorithmn to select a ***feature subset*** consisting of 3 features out of a ***featur space*** of size 10.
The input feature space consists of 10 integers: 6, 3, 1, 6, 8, 2, 3, 7, 9, 1,
and our criterion is to find a subset of size 3 in this ***feature space*** that maximizes the integer sum in this ***feature subset***.

In [5]:
```
def example_seq_backw_select():
    ex_features = [6,3,1,6,8,2,3,7,9,1]
    res_backw = seq_backw_select(features=ex_features, max_k=3,\
                                 criterion_func=simple_crit_func, print_steps=True)
    return (res_backw)

# Run example
res_backw = example_seq_backw_select()
print('\nRESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] ->', res_backw)
```

```
Exclusion from feature subset [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
exclude: 1 -> feature subset: [6, 3, 1, 6, 8, 2, 3, 7, 9]

Exclusion from feature subset [6, 3, 1, 6, 8, 2, 3, 7, 9]
exclude: 1 -> feature subset: [6, 3, 6, 8, 2, 3, 7, 9]

Exclusion from feature subset [6, 3, 6, 8, 2, 3, 7, 9]
exclude: 2 -> feature subset: [6, 3, 6, 8, 3, 7, 9]

Exclusion from feature subset [6, 3, 6, 8, 3, 7, 9]
exclude: 3 -> feature subset: [6, 3, 6, 8, 7, 9]

Exclusion from feature subset [6, 3, 6, 8, 7, 9]
exclude: 3 -> feature subset: [6, 6, 8, 7, 9]

Exclusion from feature subset [6, 6, 8, 7, 9]
exclude: 6 -> feature subset: [6, 8, 7, 9]

Exclusion from feature subset [6, 8, 7, 9]
exclude: 6 -> feature subset: [8, 7, 9]

RESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] -> [8, 7, 9]
```

Result:

The returned ***feature subset*** is similar to the result of the ***SFS*** algorithm that we have seen above, which is what we would expect: the 3 highest values (note that w defined 3 as the number of desired features for our subset) in the input feature list, since our ***criterion*** is to select the feanumerical values (= features) that yield th maximum mathematical sum in the feature subset.

4. "Plus L take away R" (+L -R)

[back to top]

The **"*Plus L take away R*" *(+L -R)*** is basically a combination of ***SFS*** and ***SBS***. It append features to the ***feature subset*** *L-times*, and afterwards it removes features *R-time* until we reach our desired size for the ***feature subset***.

Variant 1: L > R
If *L* > *R*, the algorithm starts with an empty ***feature subset*** and adds *L* features to it from the ***feature space***. Then it goes over to the next step 2, where it removes features from the ***feature subset***, after which it goes back to step 1 to add *L* features again.
Those steps are repeated until the ***feature subset** reaches the desired size *k*.

Variant 2: R > L
Else, if *R* > *L*, the algorithms starts with the whole ***feature space*** as ***feature subset***. It remove s*R* features from it before it adds back *L* features from those features tha were just removed.
Those steps are repeated until the ***feature subset** reaches the desired size *k*.

Input: the set of all features, *Y = {y_1, y_2, ..., y_d}*

- The ***+L -R*** algorithm takes the whole feature set as input, if our feature space consists of, e.g. 10, if our feature space consists of 10 dimensions (***d = 10***).

Output: a subset of features, *X_k = {x_j | j = 1, 2, ..., k; x_j ∈ Y}*, where *k = (0, 1, 2, ..., d*)

- The returned output of the algorithm is a subset of the feature space of a specified size. E.g., a subset of 5 features from a 10-dimensional feature space (***k = 5, d = 10***).

Initialization: *X_0 = Y*, *k = d*

- We initialize the algorithm with the given feature set so that the ***k = d*** (where ***k*** has the size of the feature set ***d***)

Step 1 (Inclusion):

repeat L-times:
x^+ = arg max J(x_k + x), where x ∈ Y - X_k
X_k+1 = X_k + x^+
k = k + 1
Go to Step 2

Step 2 (Exclusion):

repeat R-times:
x^- = arg max J(x_k - x), where x ∈ X_k
X_k-1 = X_k - x^-
k = k - 1
Go to Step 1

- In step 1, we go *L-times* through the ***feature space*** and look for the feature ***x^+*** which maximizes our criterion if we add it to the ***feature subset*** (where *J()* is the criterion function). Then we go over to step 2.
- In step 2, we go *R-times* through the ***feature subset*** and look for the feature ***x^-*** which minimizes our criterion if we remove it from the ***feature subset*** (where ***J()*** is the criterion function). Then we go back to step 1.
- Note that this order of steps only applies if ***L > R***, in the opposite case (***R > L***), we have to start with the ***Exlusion*** on the whole ***feature space***, followed by inclusion of removed features.

Termination: stop when ***k*** equals the number of desired features

- We add and remove features from the feature subset ***X_k*** until we reach the number of specified features for our final subset. E.g., if our desired number of features is 5 and we start with the entire feature space (*Initialization*), we would remove features from the subset until it contains 5 features.

+L -R Code

[back to top]

In [6]:
```
# Sebastian Raschka
# last updated: 03/29/2014
# "Plus L take away R" (+L -R)

from copy import deepcopy

def plus_L_minus_R(features, max_k, criterion_func, L=3, R=2, print_steps=False):
    """
    Implementation of a "Plus l take away r" algorithm.

    Keyword Arguments:
        features (list): The feature space as a list of features.
        max_k: Termination criterion; the size of the returned feature subset.
        criterion_func (function): Function that is used to evaluate the
            performance of the feature subset.
        L (int): Number of features added per iteration.
        R (int): Number of features removed per iteration.
        print_steps (bool): Prints the algorithm procedure if True.

    Returns the selected feature subset, a list of features of length max_k.

    """
    assert(L != R), 'L must be != R to avoid an infinite loop'

    ##########################
    ### +L -R for case L > R ###
    ##########################

    if L > R:
        feat_sub = []
        k = 0

        # Initialization
        while True:

            # +L (Inclusion)
            if print_steps:
                print('\nInclusion from features', features)
            for i in range(L):
                if len(features) > 0:
                    crit_func_max = criterion_func(feat_sub + [features[0]])
                    best_feat = features[0]
                    if len(features) > 1:
                        for x in features[1:]:
                            crit_func_eval = criterion_func(feat_sub + [x])
                            if crit_func_eval > crit_func_max:
                                crit_func_max = crit_func_eval
                                best_feat = x
                    features.remove(best_feat)
                    feat_sub.append(best_feat)
                    if print_steps:
                        print('include: {} -> feature_subset: {}'.format(best_feat, feat_sub))

            # -R (Exclusion)
            if print_steps:
                print('\nExclusion from feature_subset', feat_sub)
            for i in range(R):
                if len(features) + len(feat_sub) > max_k:
                    worst_feat = len(feat_sub)-1
                    worst_feat_val = feat_sub[worst_feat]
                    crit_func_max = criterion_func(feat_sub[:-1])

                    for j in reversed(range(0,len(feat_sub)-1)):
                        crit_func_eval = criterion_func(feat_sub[:j] + feat_sub[j+1:])
                        if crit_func_eval > crit_func_max:
                            worst_feat, crit_func_max = j, crit_func_eval
                            worst_feat_val = feat_sub[worst_feat]
                    del feat_sub[worst_feat]
                    if print_steps:
                        print('exclude: {} -> feature subset: {}'.format(worst_feat_val, feat_sub))


            # Termination condition
            k = len(feat_sub)
            if k == max_k:
                break

        return feat_sub

    ##########################
    ### +L -R for case L < R ###
```

```
    ############################

    else:
        # Initialization
        feat_sub = deepcopy(features)
        k = len(feat_sub)
        i = 0
        count = 0
        while True:
            count += 1
            # Exclusion step
            removed_feats = []
            if print_steps:
                print('\nExclusion from feature subset', feat_sub)
            for i in range(R):
                if len(feat_sub) > max_k:
                    worst_feat = len(feat_sub)-1
                    worst_feat_val = feat_sub[worst_feat]
                    crit_func_max = criterion_func(feat_sub[:-1])

                    for i in reversed(range(0,len(feat_sub)-1)):
                        crit_func_eval = criterion_func(feat_sub[:i] + feat_sub[i+1:])
                        if crit_func_eval > crit_func_max:
                            worst_feat, crit_func_max = i, crit_func_eval
                            worst_feat_val = feat_sub[worst_feat]
                    removed_feats.append(feat_sub.pop(worst_feat))
            if print_steps:
                print('exclude: {} -> feature subset: {}'.format(removed_feats, feat_sub))

            # +L (Inclusion)
            included_feats = []
            if len(feat_sub) != max_k:
                for i in range(L):
                    if len(removed_feats) > 0:
                        crit_func_max = criterion_func(feat_sub + [removed_feats[0]])
                        best_feat = removed_feats[0]
                        if len(removed_feats) > 1:
                            for x in removed_feats[1:]:
                                crit_func_eval = criterion_func(feat_sub + [x])
                                if crit_func_eval > crit_func_max:
                                    crit_func_max = crit_func_eval
                                    best_feat = x
                        removed_feats.remove(best_feat)
                        feat_sub.append(best_feat)
                        included_feats.append(best_feat)
                if print_steps:
                    print('\nInclusion from removed features', removed_feats)
                    print('include: {} -> feature_subset: {}'.format(included_feats, feat_sub))

            # Termination condition
            k = len(feat_sub)
            if k == max_k:
                break
            if count >= 30:
                break
        return feat_sub
```

Example +L -R:

[back to top]

Like we did for the ***SFS*** example above, let's have look at the individual steps of the ***+L -R*** algorithmn to select a ***feature subset*** consisting of 3 features out of a ***featu*** ***space*** of size 10.
Again, the input feature space consists of the 10 integers: 6, 3, 1, 6, 8, 2, 3, 7, 9, 1,
and our criterion is to find a subset of size 3 in this ***feature space*** that maximizes the integer sum in this ***feature subset***.

Example 1: L > R

[back to top]

In [7]:
```
def example_plus_L_minus_R():
    ex_features = [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
    res_plmr = plus_L_minus_R(features=ex_features, max_k=3,\
                              criterion_func=simple_crit_func, L=3, R=2, print_steps=True)

    return (res_plmr)

# Run example
res = example_plus_L_minus_R()
print('\nRESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] ->', res)
```

```
Inclusion from features [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
include: 9 -> feature_subset: [9]
include: 8 -> feature_subset: [9, 8]
include: 7 -> feature_subset: [9, 8, 7]

Exclusion from feature_subset [9, 8, 7]
exclude: 7 -> feature subset: [9, 8]
exclude: 8 -> feature subset: [9]

Inclusion from features [6, 3, 1, 6, 2, 3, 1]
include: 6 -> feature_subset: [9, 6]
include: 6 -> feature_subset: [9, 6, 6]
include: 3 -> feature_subset: [9, 6, 6, 3]

Exclusion from feature_subset [9, 6, 6, 3]
exclude: 3 -> feature subset: [9, 6, 6]
exclude: 6 -> feature subset: [9, 6]

Inclusion from features [1, 2, 3, 1]
include: 3 -> feature_subset: [9, 6, 3]
include: 2 -> feature_subset: [9, 6, 3, 2]
include: 1 -> feature_subset: [9, 6, 3, 2, 1]

Exclusion from feature_subset [9, 6, 3, 2, 1]
exclude: 1 -> feature subset: [9, 6, 3, 2]
exclude: 2 -> feature subset: [9, 6, 3]

RESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] -> [9, 6, 3]
```

Example 2: R > L

[back to top]

In [8]:
```
def example_plus_L_minus_R():
    ex_features = [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
    res_plmr = plus_L_minus_R(features=ex_features, max_k=3,\
                              criterion_func=simple_crit_func, L=2, R=3, print_steps=True)

    return (res_plmr)

# Run example
res = example_plus_L_minus_R()
print('\nRESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] ->', res)
```

```
Exclusion from feature subset [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
exclude: [1, 1, 2] -> feature subset: [6, 3, 6, 8, 3, 7, 9]

Inclusion from removed features [1]
include: [2, 1] -> feature_subset: [6, 3, 6, 8, 3, 7, 9, 2, 1]

Exclusion from feature subset [6, 3, 6, 8, 3, 7, 9, 2, 1]
exclude: [1, 2, 3] -> feature subset: [6, 3, 6, 8, 7, 9]

Inclusion from removed features [1]
include: [3, 2] -> feature_subset: [6, 3, 6, 8, 7, 9, 3, 2]

Exclusion from feature subset [6, 3, 6, 8, 7, 9, 3, 2]
exclude: [2, 3, 3] -> feature subset: [6, 6, 8, 7, 9]

Inclusion from removed features [2]
include: [3, 3] -> feature_subset: [6, 6, 8, 7, 9, 3, 3]

Exclusion from feature subset [6, 6, 8, 7, 9, 3, 3]
exclude: [3, 3, 6] -> feature subset: [6, 8, 7, 9]

Inclusion from removed features [3]
include: [6, 3] -> feature_subset: [6, 8, 7, 9, 6, 3]

Exclusion from feature subset [6, 8, 7, 9, 6, 3]
exclude: [3, 6, 6] -> feature subset: [8, 7, 9]

RESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] -> [8, 7, 9]
```

Result:

The returned ***feature subset*** really is suboptimal in this particular example for L > R (Example 1: L > R). This is due to the fact that we add multiple features to our ***feature subset*** and we also remove multiple features from it; we never add back any of the removed features to the investigated ***feature space***.

Modifying the "Plus L take away R" algorithm
This algorithm can be tweaked by adding back *r - 1* features to the ***feature subset*** after each ***Exclusion*** step to be assessed by the ***criterion function*** for inclusion in th next iteration. This is a decision that has to be made by considering the particular application, since it decreases the computational performance of the algorithm, bu improves the performance of the resulting ***feature subset*** as a classifier.

---

5. Sequential Floating Forward Selection (SFFS)

[back to top]

The ***Sequential Floating Forward Selection (SFFS)*** algorithm can be considered as extension of the simpler ***SFS*** algorithm, which we have seen in the very beginning. I constrast to ***SFFS***, the ***SFFS*** algorithm **can** remove features once they were included, so that a larger number of feature subset combinations can be sampled. It is importar to emphasize that the removal of included features is **conditional**, which makes it different from the ***+L -R*** algorithm. The ***Conditional Exclusion*** in ***SFFS*** only occurs if th resultin feature subset is assessed as "better" by the ***criterion function*** after removal of a particular feature.

Input: the set of all features, $Y = \{y_1, y_2, ..., y_d\}$

- The ***SFFS*** algorithm takes the whole feature set as input, if our feature space consists of, e.g. 10, if our feature space consists of 10 dimensions (***d = 10***).

Output: a subset of features, $X_k = \{x_j \mid j = 1, 2, ..., k; x_j \in Y\}$, where $k = (0, 1, 2, ..., d)$

- The returned output of the algorithm is a subset of the feature space of a specified size. E.g., a subset of 5 features from a 10-dimensional feature space (***k = 5, d = 10***).

Initialization: $X_0 = Y$, $k = d$

- We initialize the algorithm with an empty set ("null set") so that the ***k = 0*** (where ***k*** is the size of the subset)

Step 1 (Inclusion):

$x^+ = arg\ max\ J(x_k + x)$, where $x \in Y - X_k$
$X_{k+1} = X_k + x^+$
$k = k + 1$
Go to Step 2

Step 2 (Conditional Exclusion):

$x^- = arg\ max\ J(x_k - x)$, where $x \in X_k$
if $J(x_k - x) > J(x_k - x)$:
    $X_{k-1} = X_k - x^-$
    $k = k - 1$
Go to Step 1

- In step 1, we include the feature from the ***feature space*** that leads to the best performance increase for our ***feature subset*** (assessed by the ***criterion function***). Then, we go over to step 2
- In step 2, we only remove a feature if the resulting subset would gain an increase in performance. We go back to step 1.
- Steps 1 and 2 are reapeated until the **Termination** criterion is reached.

Termination: stop when ***k*** equals the number of desired features

---

SFFS Code

[back to top]

In [9]:
```
# Sebastian Raschka
# last updated: 03/29/2014
# Sequential Floating Forward Selection (SFFS)

def seq_float_forw_select(features, max_k, criterion_func, print_steps=False):
    """
```

```
    Implementation of Sequential Floating Forward Selection.

    Keyword Arguments:
        features (list): The feature space as a list of features.
        max_k: Termination criterion; the size of the returned feature subset.
        criterion_func (function): Function that is used to evaluate the
            performance of the feature subset.
        print_steps (bool): Prints the algorithm procedure if True.

    Returns the selected feature subset, a list of features of length max_k.

    """

    # Initialization
    feat_sub = []
    k = 0

    while True:

        # Step 1: Inclusion
        if print_steps:
            print('\nInclusion from features', features)
        if len(features) > 0:
            crit_func_max = criterion_func(feat_sub + [features[0]])
            best_feat = features[0]
            if len(features) > 1:
                for x in features[1:]:
                    crit_func_eval = criterion_func(feat_sub + [x])
                    if crit_func_eval > crit_func_max:
                        crit_func_max = crit_func_eval
                        best_feat = x
            features.remove(best_feat)
            feat_sub.append(best_feat)
            if print_steps:
                print('include: {} -> feature_subset: {}'.format(best_feat, feat_sub))

        # Step 2: Conditional Exclusion
            worst_feat_val = None
            if len(features) + len(feat_sub) > max_k:
                crit_func_max = criterion_func(feat_sub)
                for i in reversed(range(0,len(feat_sub))):
                    crit_func_eval = criterion_func(feat_sub[:i] + feat_sub[i+1:])
                    if crit_func_eval > crit_func_max:
                        worst_feat, crit_func_max = i, crit_func_eval
                        worst_feat_val = feat_sub[worst_feat]
                if worst_feat_val:
                    del feat_sub[worst_feat]
            if print_steps:
                print('exclude: {} -> feature subset: {}'.format(worst_feat_val, feat_sub))


        # Termination condition
        k = len(feat_sub)
        if k == max_k:
            break

    return feat_sub
```

Example SFFS:

[back to top]

Since the ***Exclusion*** step in the ***Sequential Floating Forward*** algorithm is ***conditional*** - a feature is only removed if the criterion function asses a better performance afte removal - we would never exclude any feature using our `simple_criterion_func()`, which returns the integer sum of a feature set. Thus, let us define another simp criterion function that we use for testing our ***SFFS*** algorithm.

Just as we did for the previous examples above, let's take a look at the individual steps of the ***SFFS*** algorithmn to select a ***feature subset*** consisting of 3 features out of ***feature space*** of size 10.
Also here, the input feature space consists of the 10 integers: 6, 3, 1, 6, 8, 2, 3, 7, 9, 1,
and our criterion is to find a subset of size 3 in this ***feature space*** that maximizes the integer sum in this ***feature subset***.

A simple criterion function with a random parameter

[back to top]

The criterion function we define below is also calculates the sum of a subset similar to the `simple_criterion_func()` we used before. However, here we add a randor integer ranging from -15 to 15 to the returned sum. Therefore, in some occasions, our criterion function can return a larger sum for a smaller subset - after we removed feature from the subset after the ***Inclusion*** step - in order to trigger the ***Conditional Exclusion*** step.

In [10]:
```
from random import randint
```

```
def simple_rand_crit_func(feat_sub):
    """
    Returns sum of numerical values of an input list plus
    a random integer ranging from -15 to 15.

    """
    return sum(feat_sub) + randint(-15,15)

# Example:
simple_rand_crit_func([1,2,4])
```

```
Out[10]: 19
```

```
In [11]: def example_seq_float_forw_select():
    ex_features = [6,3,1,6,8,2,3,7,9,1]
    res_seq_flforw = seq_float_forw_select(features=ex_features, max_k=3,\
                                criterion_func=simple_rand_crit_func, print_steps=True)
    return res_seq_flforw

# Run example
res_seq_flforw = example_seq_float_forw_select()
print('\nRESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] ->', res_seq_flforw)
```

```
Inclusion from features [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
include: 7 -> feature_subset: [7]
exclude: None -> feature subset: [7]

Inclusion from features [6, 3, 1, 6, 8, 2, 3, 9, 1]
include: 1 -> feature_subset: [7, 1]
exclude: 7 -> feature subset: [1]

Inclusion from features [6, 3, 6, 8, 2, 3, 9, 1]
include: 3 -> feature_subset: [1, 3]
exclude: None -> feature subset: [1, 3]

Inclusion from features [6, 6, 8, 2, 3, 9, 1]
include: 9 -> feature_subset: [1, 3, 9]
exclude: None -> feature subset: [1, 3, 9]

RESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] -> [1, 3, 9]
```

---

6. Sequential Floating Backward Selection (SFBS)

[back to top]

Just as in the ***SFFS*** algorithm, we have a conditional step: Here, we start with the whole feature subset and exclude features sequentially. Only if adding one of th previously excluded features back to a new ***feature subset*** improves the performance (assessed by the criterion function), we add it back in the ***Conditional Inclusio*** step.

Input: the set of all features, $Y = \{y_1, y_2, ..., y_d\}$

- The ***SFBS*** algorithm takes the whole feature set as input, if our feature space consists of, e.g. 10, if our feature space consists of 10 dimensions (***d = 10***).

Output: a subset of features, $X_k = \{x_j \mid j = 1, 2, ..., k; x_j \in Y\}$, where $k = (0, 1, 2, ..., d)$

- The returned output of the algorithm is a subset of the feature space of a specified size. E.g., a subset of 5 features from a 10-dimensional feature space (***k = 5, d = 10***).

Initialization: $X_0 = Y$, $k = d$

- We initialize the algorithm with the given feature set so that the ***k = d*** (where ***k*** has the size of the feature set ***d***)

Step 1 (Exclusion):

$x^- = arg\ max\ J(x_k - x)$, where $x \in X_k$
$X_{k-1} = X_k - x^-$
$k = k - 1$
Go to Step 2

Step 2 (Conditional Inclusion):

x^+ = arg max J(x_k + x), where x ∈ Y - X_k
if J(x_k + x) > J(x_k + x):
X_k+1 = X_k + x^+
k = k + 1
Go to Step 1

- In step 1, we exclude the feature from the ***feature space*** that yields the best performance increase of the ***feature subset*** (assessed by the ***criterion function***). Then, we go over to step 2.
- In step 2, we only include one of the removed features if the resulting subset would gain an increase in performance. We go back to step 1.
- Steps 1 and 2 are reapeated until the **Termination** criterion is reached.

SFBS Code

[back to top]

In [16]:
```
# Sebastian Raschka
# last updated: 03/29/2014
# Sequential Floating Backward Selection (SFBS)

from copy import deepcopy

def seq_float_backw_select(features, max_k, criterion_func, print_steps=False):
    """
    Implementation of Sequential Floating Backward Selection.

    Keyword Arguments:
        features (list): The feature space as a list of features.
        max_k: Termination criterion; the size of the returned feature subset.
        criterion_func (function): Function that is used to evaluate the
            performance of the feature subset.
        print_steps (bool): Prints the algorithm procedure if True.

    Returns the selected feature subset, a list of features of length max_k.

    """

    # Initialization
    feat_sub = deepcopy(features)
    k = len(feat_sub)
    i = 0
    excluded_features = []

    while True:

        # Termination condition
        k = len(feat_sub)
        if k == max_k:
            break

        # Step 1: Exclusion
        if print_steps:
            print('\nExclusion from feature subset', feat_sub)
        worst_feat = len(feat_sub)-1
        worst_feat_val = feat_sub[worst_feat]
        crit_func_max = criterion_func(feat_sub[:-1])

        for i in reversed(range(0,len(feat_sub)-1)):
            crit_func_eval = criterion_func(feat_sub[:i] + feat_sub[i+1:])
            if crit_func_eval > crit_func_max:
                worst_feat, crit_func_max = i, crit_func_eval
                worst_feat_val = feat_sub[worst_feat]
        excluded_features.append(feat_sub[worst_feat])
        del feat_sub[worst_feat]
        if print_steps:
            print('exclude: {} -> feature subset: {}'.format(worst_feat_val, feat_sub))

        # Step 2: Conditional Inclusion
        if len(excluded_features) > 0 and len(feat_sub) != max_k:
            best_feat = None
            best_feat_val = None
            crit_func_max = criterion_func(feat_sub)
            for i in range(len(excluded_features)):
                crit_func_eval = criterion_func(feat_sub + [excluded_features[i]])
                if crit_func_eval > crit_func_max:
                    best_feat, crit_func_max = i, crit_func_eval
                    best_feat_val = excluded_features[best_feat]
            if best_feat:
```

```
                feat_sub.append(excluded_features[best_feat])
                del excluded_features[best_feat]
            if print_steps:
                    print('include: {} -> feature subset: {}'.\
                          format(best_feat_val, feat_sub))

    return feat_sub
```

Example SFBS:

[back to top]

Note that the ***Inclusion*** step in the ***Sequential Floating Backward Selection*** algorithm is ***conditional*** - a feature is only added back if the criterion function asses a bette performance after its inclusion in the ***feature subset***.
Therefore, we have to be a little bit careful about the ***criterion function***: If we used our `simple_criterion_func()`, which returns the integer sum of a subset, we woul trigger an infinite loop and never reach the termination criterion - assuming that our feature space consists of positive integers. The reason is that th `simple_criterion_func()` would always return a larger sum if we include a positive integer (our feature) to the ***feature subset***, thus let us use our second simpl criterion function, which contains a (pseudo) random integer between -15 and 15 to the returned sum.
In order to reduce the number of iterations, we set the number of desired features for the ***feature subset*** (via the argument `max_k`) to 7.

In [18]:
```
def example_seq_float_backw_select():
    ex_features = [6,3,1,6,8,2,3,7,9,1]
    res_seq_flbackw = seq_float_backw_select(features=ex_features, max_k=7,\
                                criterion_func=simple_rand_crit_func, print_steps=True)
    return res_seq_flbackw

# Run example
res_seq_flbackw = example_seq_float_backw_select()
print('\nRESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] ->', res_seq_flbackw)
```

```
Exclusion from feature subset [6, 3, 1, 6, 8, 2, 3, 7, 9, 1]
exclude: 2 -> feature subset: [6, 3, 1, 6, 8, 3, 7, 9, 1]
include: None -> feature subset: [6, 3, 1, 6, 8, 3, 7, 9, 1]

Exclusion from feature subset [6, 3, 1, 6, 8, 3, 7, 9, 1]
exclude: 3 -> feature subset: [6, 3, 1, 6, 8, 7, 9, 1]
include: 3 -> feature subset: [6, 3, 1, 6, 8, 7, 9, 1, 3]

Exclusion from feature subset [6, 3, 1, 6, 8, 7, 9, 1, 3]
exclude: 3 -> feature subset: [6, 3, 1, 6, 8, 7, 9, 1]
include: None -> feature subset: [6, 3, 1, 6, 8, 7, 9, 1]

Exclusion from feature subset [6, 3, 1, 6, 8, 7, 9, 1]
exclude: 6 -> feature subset: [3, 1, 6, 8, 7, 9, 1]

RESULT: [6, 3, 1, 6, 8, 2, 3, 7, 9, 1] -> [3, 1, 6, 8, 7, 9, 1]
```

7. Genetic Algorithm (GA)

to be continued ...

In [17]:

In []: