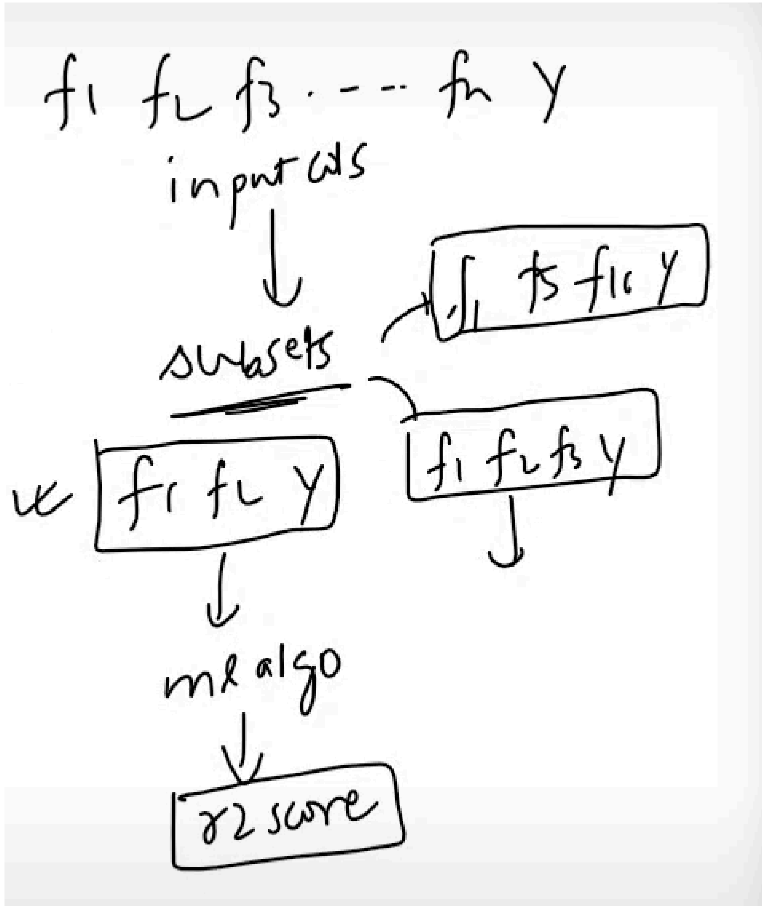
Feature Selection (Wrapper Methods)

Flaw of Filter Methods

- They are not able to study the relationship between 2 features
- Wrapper method solves this issue

How They Work:

- **Subset Evaluation:** The method selects a candidate subset of features and trains a model using only those features.
- **Performance Measurement:**The model's performance (e.g., accuracy, MSE) is measured, and this score is used to determine if the feature subset is good.



- It generates subsets like → F1, F2 & Y F1,F2,F3&Y
-

If n = 3, it will try the following subsets:

1. Feature 1
2. Feature 2
3. Feature 3
4. Feature 1 + Feature 2
5. Feature 1 + Feature 3
6. Feature 2 + Feature 3
7. Feature 1 + Feature 2 + Feature 3

- Apply a ML Algo like Linear Regression on each subset

- & calculate R2 score,Accuracy,MSE, etc
- More the score, important the features

✨Search Strategy (IMP for Interview)

Common strategies include:

- **Exhaustive Feature Selection/Best Subset Selection**
- **Forward Selection:** Start with no features, add one at a time.
    - Stop when no further improvement is observed.
- **Backward Elimination:** Start with all features, remove one at a time.
    - Stop when performance degrades significantly.
- **Recursive Feature Elimination (RFE):** Rank features by importance and remove the least important iteratively.

Exhaustive Feature Selection/Best Subset Selection

- You try out **all the possible subsets** & apply ML Algo on em
- Select the best ones
- **Works well till 10 columns**

Disadvantages

- Computational Complexity
    - You have to train $2^n - 1$ models
- Risk of Overfitting
- Requires a Good Evaluation Metric

Install mlxtend

`!pip install --upgrade scikit-learn mlxtend`

Used for:

- **Model Evaluation:** Tools for assessing model performance beyond standard metrics (e.g., plotting decision regions, learning curves).
- **Feature Selection & Extraction:** Techniques to choose the most relevant features or create new ones (e.g., sequential feature selection).
- **Data Preprocessing:** Helpful preprocessing utilities.
- **Ensemble Methods:** Implementations of ensemble learning algorithms.
- **General Utilities:** Various helper functions for common ML tasks.

We will use IRIS Dataset

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.model_selection import cross_val_score
import pandas as pd

!pip install --upgrade scikit-learn mlxtend
```

```
df = pd.read_csv('https://gist.githubusercontent.com/curran/a08a1080b88344b0c8a7/raw/0e7a9b0a5d22642a06d3d5b9bcbad9890c8ee534/iris.csv')
df.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS

lr = LogisticRegression()

sel = EFS(lr, max_features=4, scoring='accuracy', cv=5)
```

`lr` → **LogisticRegression**

`max_features=4` → we want 4 feature at most

You can also give `min_features=1`

`cv=5` → 5-fold cross-validation

`scoring`: This is the evaluation metric used to assess the performance of the feature subsets. In the example, **accuracy** is used, meaning the model's accuracy will be evaluated for each feature subset.

`'accuracy'` is a common metric for **classification** problems

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Metric	Use Case
`'accuracy'`	Default for balanced datasets (equal class distribution).
`'precision'`	Focus on minimizing **false positives** (e.g., spam detection).
`'recall'`	Focus on minimizing **false negatives** (e.g., cancer diagnosis).
`'f1'`	Balance precision and recall (imbalanced datasets).
`'roc_auc'`	Evaluate ranking performance (e.g., credit scoring).
`'balanced_accuracy'`	Accuracy adjusted for class imbalance.

Most Famous/Commonly Used Metrics

1. `'accuracy'`: Simplest and most intuitive for balanced datasets.
2. `'f1'`: Best for imbalanced datasets (e.g., fraud detection).

3. `'roc_auc'`: Preferred when probabilistic rankings matter (e.g., medical testing).

`cv`: This stands for **cross-validation**. It defines how many times the dataset will be split into training and validation sets to assess the performance.

In the example, **5-fold cross-validation** is used, meaning the data will be divided into 5 subsets, and each subset will be used for validation while the others are used for training.

In short, you do train-test-split 5 times & measure accuracy and find the mean.

```
model = sel.fit(df.iloc[:,:4],df['species'])
```

- 👆 `sel.fit(X,y)`

`df.iloc[:,:4]`:

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

`df['species']`:

```
0          setosa
1          setosa
2          setosa
3          setosa
4          setosa
          ...
145    virginica
146    virginica
147    virginica
148    virginica
149    virginica
```

```
model.best_score_

Output: 0.9733333333333334
```

- Meaning: There is 1 subset whose accuracy score is 97%
- Find out which?

```
model.best_feature_names_

Output:
('sepal_length', 'sepal_width', 'petal_length', 'petal_width')
```

See all the subsets & their results:

```
# detailed output
model.subsets_
```

```
{0: {'feature_idx': (0,),
  'cv_scores': array([0.66666667, 0.73333333, 0.76666667, 0.76666667, 0.83333333]),
  'avg_score': 0.7533333333333333,
  'feature_names': ('sepal_length',)},
 1: {'feature_idx': (1,),
  'cv_scores': array([0.53333333, 0.56666667, 0.53333333, 0.53333333, 0.63333333]),
  'avg_score': 0.5599999999999999,
  'feature_names': ('sepal_width',)},
 2: {'feature_idx': (2,),
  'cv_scores': array([0.93333333, 1.        , 0.9       , 0.93333333, 1.        ]),
  'avg_score': 0.9533333333333334,
  'feature_names': ('petal_length',)},
 3: {'feature_idx': (3,),
  'cv_scores': array([1.        , 0.96666667, 0.9       , 0.93333333, 1.        ]),
  'avg_score': 0.96,
  'feature_names': ('petal_width',)},
 4: {'feature_idx': (0, 1),
  'cv_scores': array([0.73333333, 0.83333333, 0.76666667, 0.86666667, 0.9       ]),
  'avg_score': 0.8200000000000001,
  'feature_names': ('sepal_length', 'sepal_width')},
 5: {'feature_idx': (0, 2),
  'cv_scores': array([0.93333333, 1.        , 0.9       , 0.93333333, 1.        ]),
  'avg_score': 0.9533333333333334,
  'feature_names': ('sepal_length', 'petal_length')},
 6: {'feature_idx': (0, 3),
...
  'avg_score': 0.9733333333333334,
  'feature_names': ('sepal_length',
   'sepal_width',
   'petal_length',
   'petal_width')}}
```

- Convert this into df

```
metric_df = pd.DataFrame.from_dict(model.get_metric_dict()).T
metric_df
```

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err
0	(0,)	[0.6666666666666666, 0.7333333333333333, 0.766...	0.753333	(sepal_length,)	0.069612	0.05416	0.02708
1	(1,)	[0.5333333333333333, 0.5666666666666667, 0.533...	0.56	(sepal_width,)	0.049963	0.038873	0.019437
2	(2,)	[0.9333333333333333, 1.0, 0.9, 0.9333333333333...	0.953333	(petal_length,)	0.051412	0.04	0.02
3	(3,)	[1.0, 0.9666666666666667, 0.9, 0.9333333333333...	0.96	(petal_width,)	0.049963	0.038873	0.019437
4	(0, 1)	[0.7333333333333333, 0.8333333333333334, 0.766...	0.82	(sepal_length, sepal_width)	0.079462	0.061824	0.030912
5	(0, 2)	[0.9333333333333333, 1.0, 0.9, 0.9333333333333...	0.953333	(sepal_length, petal_length)	0.051412	0.04	0.02
6	(0, 3)	[0.9333333333333333, 0.9666666666666667, 0.933...	0.953333	(sepal_length, petal_width)	0.034274	0.026667	0.013333
7	(1, 2)	[0.9333333333333333, 1.0, 0.9, 0.9333333333333...	0.953333	(sepal_width, petal_length)	0.051412	0.04	0.02
8	(1, 3)	[0.9333333333333333, 0.9666666666666667, 0.9, ...	0.94	(sepal_width, petal_width)	0.032061	0.024944	0.012472
9	(2, 3)	[0.9666666666666667, 0.9666666666666667, 0.933...	0.96	(petal_length, petal_width)	0.032061	0.024944	0.012472
10	(0, 1, 2)	[0.9333333333333333, 1.0, 0.9, 0.9333333333333...	0.953333	(sepal_length, sepal_width, petal_length)	0.051412	0.04	0.02
11	(0, 1, 3)	[0.9, 0.9666666666666667, 0.9333333333333333, ...	0.946667	(sepal_length, sepal_width, petal_width)	0.043691	0.033993	0.016997
12	(0, 2, 3)	[0.9666666666666667, 0.9666666666666667, 0.933...	0.966667	(sepal_length, petal_length, petal_width)	0.027096	0.021082	0.010541
13	(1, 2, 3)	[0.9666666666666667, 0.9666666666666667, 0.933...	0.966667	(sepal_width, petal_length, petal_width)	0.027096	0.021082	0.010541
14	(0, 1, 2, 3)	[0.9666666666666667, 1.0, 0.9333333333333333, ...	0.973333	(sepal_length, sepal_width, petal_length, peta...	0.032061	0.024944	0.012472

`pd.DataFrame.from_dict()` takes the **dictionary** (output of `model.get_metric_dict()`) and **converts it into a → pandas DataFrame**.

> **dictionary → pandas DataFrame.**

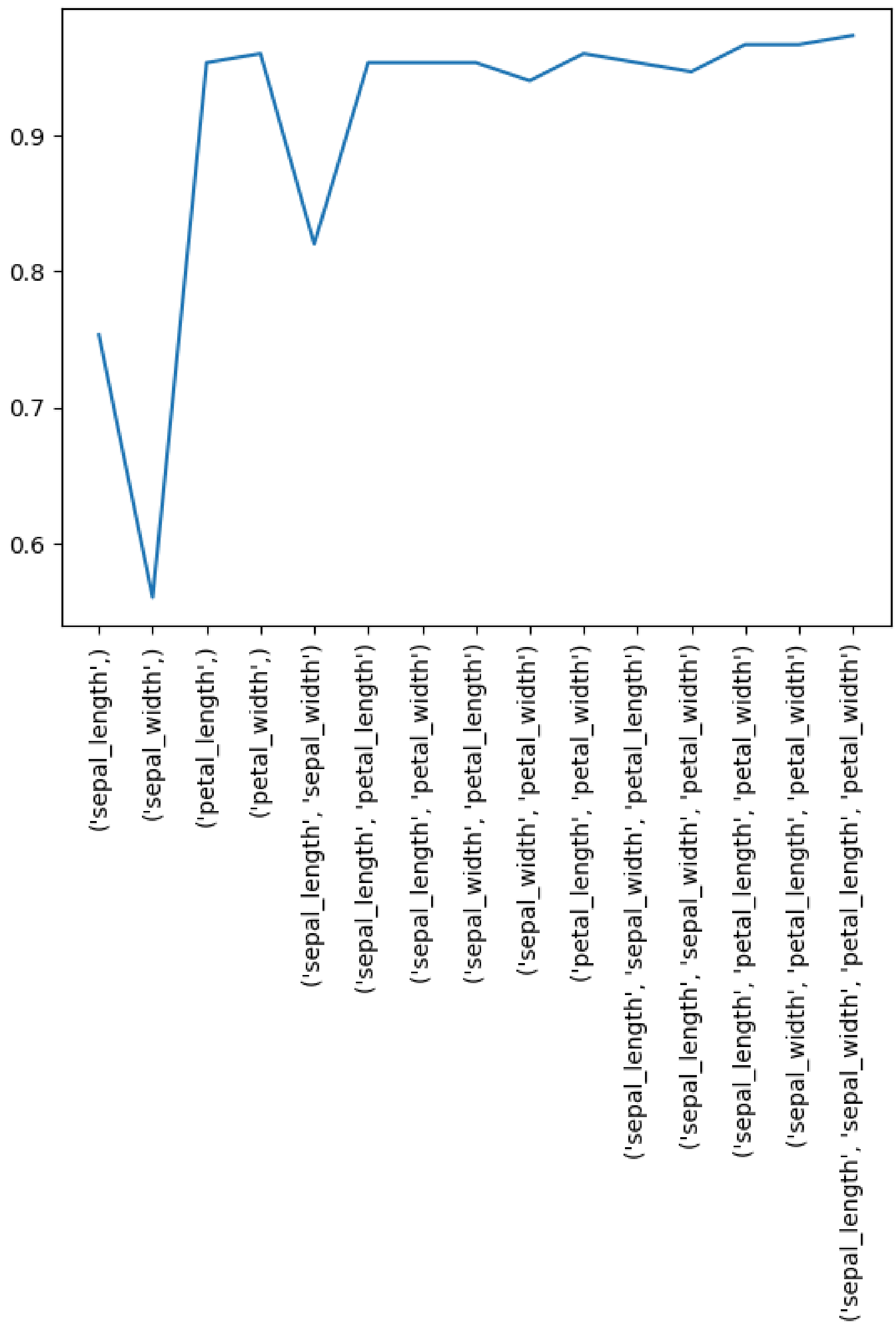
`from_dict()` is a **pandas** function used to create a DataFrame from a dictionary.

`model.get_metric_dict()` → returns a dictionary

```
{0: {'feature_idx': (0,),
  'cv_scores': array([0.66666667, 0.73333333, 0.76666667, 0.76666667, 0.83333333]),
  'avg_score': 0.7533333333333333,
  'feature_names': ('sepal_length',),
  'ci_bound': 0.0696116851832301,
  'std_dev': 0.054160256030906434,
  'std_err': 0.027080128015453217},
 1: {'feature_idx': (1,),
  'cv_scores': array([0.53333333, 0.56666667, 0.53333333, 0.53333333, 0.63333333]),
  'avg_score': 0.5599999999999999,
  'feature_names': ('sepal_width',),
  'ci_bound': 0.04996313008452825,
  'std_dev': 0.038873012632301994,
  'std_err': 0.019436506316150997},
 2: {'feature_idx': (2,),
  'cv_scores': array([0.93333333, 1.        , 0.9       , 0.93333333, 1.        ]),
  'avg_score': 0.9533333333333334,
  'feature_names': ('petal_length',),
  'ci_bound': 0.05141163671272628,
  'std_dev': 0.039999999999999994,
  'std_err': 0.019999999999999997}
```

Plot `features` vs `avg_score`

```
import matplotlib.pyplot as plt

plt.plot([str(k) for k in metric_df['feature_names']],metric_df['avg_score'])
plt.xticks(rotation=90)
plt.show()
```



`[str(k) for k in metric_df['feature_names']]` : **list comprehension**

- It iterates over each element (`k`) in `metric_df['feature_names']` and converts each element into a string.

- **Converting to string** ensures that if the feature names are in any non-string format (like integers), they will be properly displayed as labels on the x-axis.

`metric_df['avg_score']` : This is the **y-axis** data

Alternative

- `.astype(str)` / `.map(str)` / . `apply(str)` instead of **list comprehension**

```
import matplotlib.pyplot as plt

plt.plot(metric_df['feature_names'].astype(str),metric_df['avg_score'])
plt.xticks(rotation=90)
plt.show()
```

```
import matplotlib.pyplot as plt

plt.plot(metric_df['feature_names'].map(str),metric_df['avg_score'])
plt.xticks(rotation=90)
plt.show()
```

```
import matplotlib.pyplot as plt

plt.plot(metric_df['feature_names'].apply(str),metric_df['avg_score'])
plt.xticks(rotation=90)
plt.show()
```

Regression Example

```
df = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/maste
r/BostonHousing.csv')
df.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(df.iloc[:,:-1], df['medv'], test_size=0.2, random_state=1)
```

- `medv` → **y**

```
print(X_train.shape)
print(X_test.shape)

Output:
print(X_train.shape)
print(X_test.shape)
```

Before applying any feature selection

- Scale the data with std scaler

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

- `StandardScaler` →This transforms the data so that it has a **mean of 0** and a **standard deviation of 1**

Baseline Model:

```
# baseline model
import numpy as np
from sklearn.metrics import r2_score
model = LinearRegression()

print("training",np.mean(cross_val_score(model, X_train, y_train, cv=5, scoring
='r2')))
print("testing",np.mean(cross_val_score(model, X_test, y_test, cv=5, scoring
='r2')))

Output:
training 0.7025123301096213
testing 0.6514899901155404
```

Ex.

`model =` **`LogisticRegression`** `()`

`sel = EFS(model, max_features=4, scoring='accuracy', cv=5)`

- Instead of `EFS`, we are evaluating **`cross_val_score`**

`cross_val_score()`:

- This function from **`sklearn.model_selection`** is used to perform **cross-validation** on a given model.
- It splits your data into **`cv=5`** (5) **folds** (subsets of your data) and trains/evaluates the model on different subsets during each fold.
- It computes the model's performance for each fold (using a scoring metric), so you can get an understanding of how well the model is likely to perform on new data.

```
cross_val_score(model, X_train, y_train, cv=5, scoring='r2')

Output:
array([0.75350272, 0.69202385, 0.68225547, 0.66901198, 0.71576764])
```

Now use EFS:

```
lr = LinearRegression()

exh = EFS(lr, max_features=13, scoring='r2', cv=10, print_progress=True,n_jobs=-1)

sel = exh.fit(X_train, y_train)

Output:
Features: 8191/8191
```

`n_jobs=-1` → Job gets divided into multiple cores.

Makes the operation faster

`cv=10`:

- This specifies the number of **cross-validation folds** to use. In this case, **10-fold cross-validation** is used, meaning the data is split into 10 subsets, and the model is trained and evaluated 10 times, each time using a different subset for testing and the rest for training.

`print_progress=True`:

- This enables **progress printing** during the feature selection process. It will show the current status of the feature selection, such as the number of features tested or the current best feature set.

> 💡 👆 **This took 40 seconds to run.**

```
sel.best_score_

Output:
0.6827988156800063
```

```
sel.best_feature_names_

Output:
('0', '1', '4', '5', '7', '8', '9', '10', '11', '12')
```

- We’re not getting column names as our df has been converted into NumPy array.

```
metric_df = pd.DataFrame.from_dict(sel.get_metric_dict()).T
metric_df
```

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err
0	(0,)	[0.03941987916919132, 0.12695789031653215, -0....	0.129009	(0,)	0.064226	0.086475	0.028825
1	(1,)	[0.14236716209182754, -0.10598329567838705, 0....	0.100963	(1,)	0.076751	0.103339	0.034446
2	(2,)	[0.4055276765549376, 0.0029283993633670846, -0...	0.210465	(2,)	0.139709	0.188107	0.062702
3	(3,)	[-0.07110886674980432, -0.08269807310551558, 0....	-0.025663	(3,)	0.055426	0.074627	0.024876
4	(4,)	[0.18869831316675, 0.03113193162308736, 0.0348...	0.17746	(4,)	0.114827	0.154605	0.051535
...	...	...	...	...	...	...	...
8186	(0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.8791441026861514, 0.576226384714265, 0.4354...	0.679213	(0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.096133	0.129436	0.043145
8187	(0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.8731761731752511, 0.5384374917854684, 0.458...	0.679018	(0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.098986	0.133276	0.044425
8188	(0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.8494627278072393, 0.5454006150975994, 0.444...	0.66547	(0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.099036	0.133343	0.044448
8189	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.8358451877451422, 0.5448662375728606, 0.449...	0.670075	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.094633	0.127415	0.042472
8190	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.8674059553865395, 0.5385295808629285, 0.458...	0.677417	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.098114	0.132102	0.044034

Calculated adjusted r2 score

Formula:

$$R^2_{\text{adj}} = 1 - \left(\frac{(1 - R^2)(n - 1)}{n - p - 1} \right)$$

Where:

- n is the number of data points (samples).
- p is the number of predictors (features) in the model.

can also be written as:

$$\text{Adjusted } R^2 = 1 - (1 - R^2) \times \frac{n - 1}{n - p - 1}$$

```
def adjust_r2(r2, num_examples, num_features):
    coef = (num_examples - 1) / (num_examples - num_features - 1)
    return 1 - (1 - r2) * coef
```

- `r2` : The original R^2 score (e.g., 0.85).
- `num_examples` : Number of data points (**rows**) in your dataset (e.g., 404).
- `num_features` : Number of features (**columns**) used in the model.

```
metric_df['observations'] = 404
metric_df['num_features'] = metric_df['feature_idx'].apply(lambda x:len(x))
metric_df['adjusted_r2'] = adjust_r2(metric_df['avg_score'],metric_df['observa
tions'],metric_df['num_features'])
```

- Why 404?
  - `print(X_train.shape)` → (**404**, 13)

`metric_df['feature_idx'].apply(lambda x: len(x))` :

`.apply(lambda x: len(x))` :

- `.apply()` is a method that applies a function along each element of the specified column (`'feature_idx'` in this case).
- `lambda x: len(x)` is a **lambda function** that:
  - Takes each element `x` in the `'feature_idx'` column
  - `len(x)` calculates the length of `x`, which is the number of feature indices in that list or array.

```
metric_df.sort_values('adjusted_r2',ascending=False)
```

	len(feature_idx)	cv_scores	avg_score R2	feature_names	ci_bound	std_dev	std_err	observations	num_features	adjusted_r2
7975	(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)	[0.8855189158291968, 0.5742220049707852, 0.437...	0.682799	(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)	0.096995	0.130595	0.043532	404	10	0.674728
7408	(0, 1, 4, 5, 7, 8, 9, 10, 12)	[0.8717831363927702, 0.5819307800982585, 0.462...	0.680483	(0, 1, 4, 5, 7, 8, 9, 10, 12)	0.090811	0.122269	0.040756	404	9	0.673185
8141	(0, 1, 2, 4, 5, 7, 8, 9, 10, 11, 12)	[0.8792702841985806, 0.5752245789381261, 0.438...	0.681125	(0, 1, 2, 4, 5, 7, 8, 9, 10, 11, 12)	0.096068	0.129348	0.043116	404	11	0.672177
8150	(0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12)	[0.8734082301119797, 0.538138251576179, 0.4610...	0.680994	(0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12)	0.098795	0.133019	0.04434	404	11	0.672043
8153	(0, 1, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.8853169531726776, 0.5751761822045902, 0.434...	0.680914	(0, 1, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.097075	0.130703	0.043568	404	11	0.67196
...	...	...	...	...	...	...	...	...	...	...
53	(3, 11)	[0.07227421305699011, -0.026141441832760126, 0...	0.073485	(3, 11)	0.069934	0.09416	0.031387	404	2	0.068864
11	(11,)	[0.1200629474726852, 0.03143835749752166, -0.0...	0.068712	(11,)	0.071116	0.095752	0.031917	404	1	0.066396

- We sorted the df by adjusted r2 score, so that the best results show on top.

Now transform the `X_train` & `X_test`

```
X_train_sel = sel.transform(X_train)
X_test_sel = sel.transform(X_test)
```

- Now these have 10 columns. Not 13
- `transform()` will remove the columns and keep the best 10 columns

Train the

💡 REMEMBER the previous code:

`exh = EFS(lr, max_features=13, scoring='r2', cv=10, print_progress=True,n_jobs=-1)`

`sel =` **`exh.fit(X_train, y_train)`**

- It has given us 10 columns based on r2 score

Create another LR model & calculate the average r2 score with cross_val_score

```
model = LinearRegression()

print("training",np.mean(cross_val_score(model, X_train_sel, y_train, cv=5, scorin
print("testing",np.mean(cross_val_score(model, X_test_sel, y_test, cv=5, scoring=
```

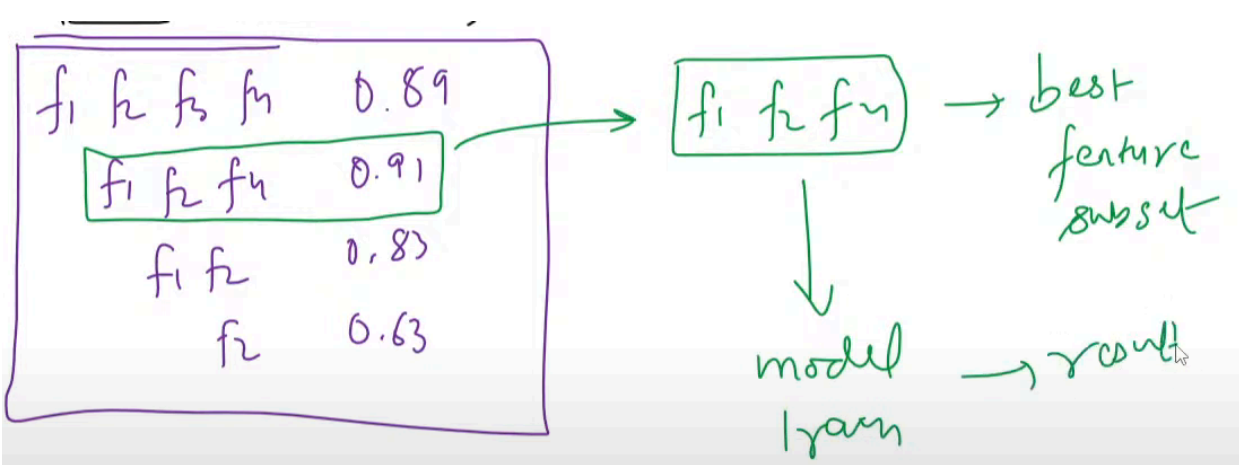
Output:

training 0.7100327839218562
testing 0.7205819296124483

- Overfitting is being removed

Sequential Backward Elimination/Selection

- Start with all features, remove one at a time.
- But which column to eliminate?
    - It removes 1 column at a time and checks results
- Will keep the model with best results
- Repeat this process
- **Iterations= No. of features**



- **Faster**

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

# load the dataset
data = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/master

# separate the target variable
X = data.drop("medv", axis=1)
y = data['medv']

# split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

print(X_train.shape)

Output:
(404, 13)
```

Scale:

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Apply LR Model:

```
model = LinearRegression()

print("training",np.mean(cross_val_score(model, X_train, y_train, cv=5, scoring='r
print("testing",np.mean(cross_val_score(model, X_test, y_test, cv=5, scoring='r2'

Output:
training 0.7025123301096213
testing 0.6514899901155404
```

Apply SFS:

```
lr = LinearRegression()

# perform backward elimination
sfs = SFS(lr, k_features='best', forward=False, floating=False, scoring='r2',cv=5)

sfs.fit(X_train, y_train)
```

`k_features='best'` → How many features you want?

- `'best'` → Unspecified. Just give me best output

`k_features='5'` → Give me 5 best features

`forward=False` → For forward selection, select True

`floating=False` → floating is a variation of this. we don't want that rn

- 👆 **Took 0.7 sec against 40 sec for EFS**

```
sfs.k_feature_idx_
```

```
Output:
(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)
```

- Gave us the same result as EFS.

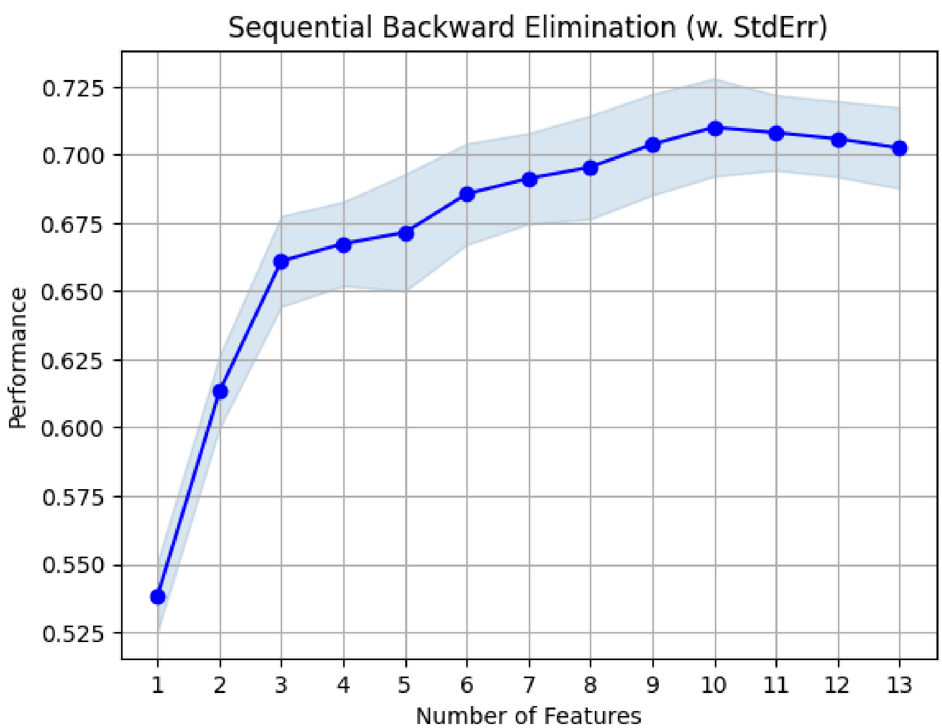
- Adjusted r2:

```
metric_df = pd.DataFrame.from_dict(sfs.get_metric_dict()).T

metric_df['observations'] = 404
metric_df['num_features'] = metric_df['feature_idx'].apply(lambda x:len(x))
metric_df['adjusted_r2'] = adjust_r2(metric_df['avg_score'],metric_df['observations'],metric_df['num_features'])

metric_df
```

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err	observations	num_features	adjusted_r2
13	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.7535027170817178, 0.6920238509138777, 0.682...	0.702512	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.038207	0.029727	0.014863	404	13	0.692596
12	(0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12)	[0.7532855958710695, 0.6944570477695307, 0.693...	0.70581	(0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12)	0.035641	0.02773	0.013865	404	12	0.696781
11	(0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12)	[0.754710892556849, 0.6959627893665097, 0.7017...	0.708109	(0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12)	0.035367	0.027516	0.013758	404	11	0.699918
10	(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)	[0.7769593921905563, 0.6884741223718953, 0.702...	0.710033	(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)	0.046075	0.035848	0.017924	404	10	0.702654
9	(0, 1, 4, 5, 7, 8, 9, 10, 12)	[0.7706104220711025, 0.6854023389684323, 0.690...	0.704324	(0, 1, 4, 5, 7, 8, 9, 10, 12)	0.046449	0.036139	0.018069	404	9	0.69757
8	(0, 1, 4, 5, 7, 8, 10, 12)	[0.7681719744800458, 0.6822126526818693, 0.670...	0.697727	(0, 1, 4, 5, 7, 8, 10, 12)	0.04882	0.037984	0.018992	404	8	0.691605
7	(0, 1, 4, 5, 7, 10, 12)	[0.7671638009750725, 0.6812300799626649, 0.661...	0.692234	(0, 1, 4, 5, 7, 10, 12)	0.051644	0.040181	0.02009	404	7	0.686794
6	(1, 4, 5, 7, 10, 12)	[0.7519120213497091, 0.6756087674652564, 0.646...	0.686004	(1, 4, 5, 7, 10, 12)	0.046845	0.036447	0.018224	404	6	0.681258
5	(4, 5, 7, 10, 12)	[0.7525552802357769, 0.6665033988504306, 0.639...	0.681065	(4, 5, 7, 10, 12)	0.051233	0.039861	0.019931	404	5	0.677058
4	(5, 7, 10, 12)	[0.7384743962575444, 0.640118850766883, 0.5873...	0.662544	(5, 7, 10, 12)	0.063384	0.049315	0.024658	404	4	0.659161
3	(5, 10, 12)	[0.7215896884753017, 0.6288372046797153, 0.633...	0.661012	(5, 10, 12)	0.04259	0.033136	0.016568	404	3	0.65847
2	(5, 12)	[0.6330856272904802, 0.5779812120755249, 0.586...	0.613259	(5, 12)	0.034066	0.026505	0.013252	404	2	0.61133
1	(12,)	[0.5472998394577442, 0.49002001493399727, 0.53...	0.538451	(12,)	0.032755	0.025485	0.012742	404	1	0.537303

Plot Graph:

```
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs


fig1 = plot_sfs(sfs.get_metric_dict(), kind='std_err',)

plt.title('Sequential Backward Elimination (w. StdErr)')
plt.grid()
plt.show()
```



- `from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs`:
  - This imports the **plotting function** for visualizing the results of **sequential feature selection** from the `mlxtend` library.
- `sfs.get_metric_dict()`:
  - This retrieves the metric dictionary from the **sequential feature selection** (SFS) object, which contains performance metrics for each subset of selected features.

`plot_sfs(sfs.get_metric_dict(), kind='std_err')` :

- This function plots the sequential feature selection results. The `kind='std_err'` argument specifies that the **standard error** of the performance metric should be shown on the plot.
- **Light blue region** = Std error

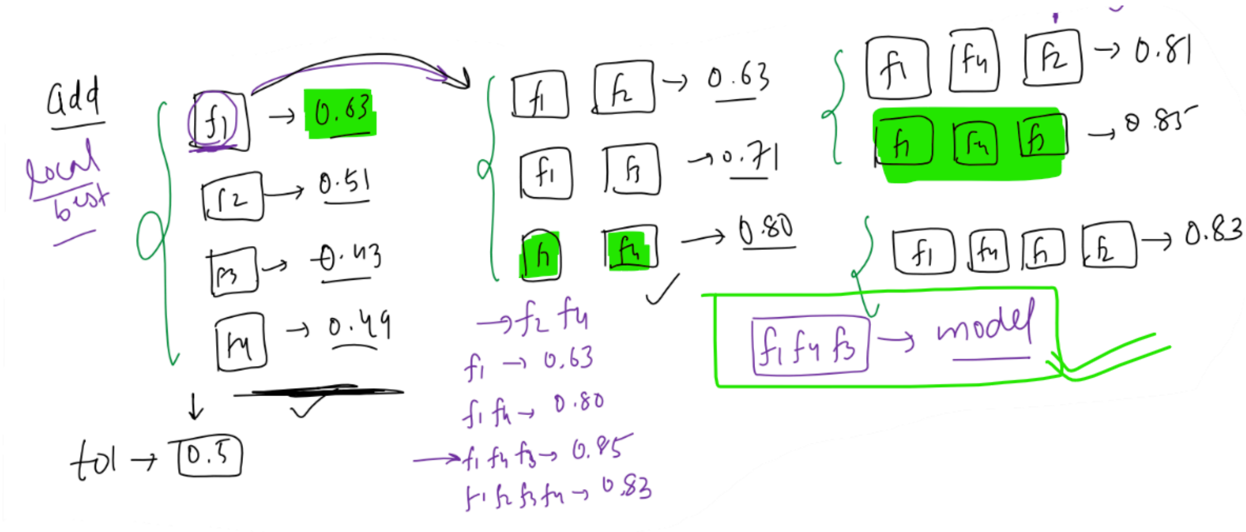
```
X_train_sel = sfs.transform(X_train)
X_test_sel = sfs.transform(X_test)

model = LinearRegression()

print("training",np.mean(cross_val_score(model, X_train_sel, y_train, cv=5, sco
ring='r2')))
print("testing",np.mean(cross_val_score(model, X_test_sel, y_test, cv=5, scorin
g='r2')))

Output:
training 0.7100327839218562
testing 0.7205819296124483
```

Disadvantage:

- You can miss the best.

Forward selection

- Opposite of Backward Elimination
- Start with no features, add one at a time.



No. of models:

$$\frac{n(n+1)}{2}$$

n → Features

Code

- Everything is same as above code
- Just change `forward=True`

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

lr = LinearRegression()

# perform backward elimination
sfs = SFS(lr, k_features='best', forward=True, floating=False, scoring='r2',cv=
5)
```

```
sfs.fit(X_train, y_train)

metric_df = pd.DataFrame.from_dict(sfs.get_metric_dict()).T

metric_df['observations'] = 404
metric_df['num_features'] = metric_df['feature_idx'].apply(lambda x:len(x))
metric_df['adjusted_r2'] = adjust_r2(metric_df['avg_score'],metric_df['observa
tions'],metric_df['num_features'])

metric_df
```

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err	observations	num_features	adjusted_r2
1	(12,)	[0.5472998394577442, 0.49002001493399727, 0.53...	0.538451	(12,)	0.032755	0.025485	0.012742	404	1	0.537303
2	(5, 12)	[0.6330856272904802, 0.5779812120755249, 0.586...	0.613259	(5, 12)	0.034066	0.026505	0.013252	404	2	0.61133
3	(5, 10, 12)	[0.7215896884753017, 0.6288372046797153, 0.633...	0.661012	(5, 10, 12)	0.04259	0.033136	0.016568	404	3	0.65847
4	(5, 10, 11, 12)	[0.725877216548624, 0.6342604286872173, 0.6558...	0.667383	(5, 10, 11, 12)	0.039611	0.030819	0.01541	404	4	0.664048
5	(5, 7, 10, 11, 12)	[0.7440756174774326, 0.6473449858158777, 0.614...	0.671496	(5, 7, 10, 11, 12)	0.055057	0.042836	0.021418	404	5	0.667369
6	(4, 5, 7, 10, 11, 12)	[0.7554472741494642, 0.6695521143038103, 0.653...	0.68562	(4, 5, 7, 10, 11, 12)	0.047626	0.037054	0.018527	404	6	0.680869
7	(1, 4, 5, 7, 10, 11, 12)	[0.7548843433907461, 0.6798073590310516, 0.662...	0.6913	(1, 4, 5, 7, 10, 11, 12)	0.042762	0.03327	0.016635	404	7	0.685844
8	(0, 1, 4, 5, 7, 10, 11, 12)	[0.7678201537729974, 0.6827004362205138, 0.671...	0.695442	(0, 1, 4, 5, 7, 10, 11, 12)	0.048774	0.037948	0.018974	404	8	0.689274
9	(0, 1, 4, 5, 7, 8, 10, 11, 12)	[0.7746648196407366, 0.6850585456872307, 0.683...	0.703763	(0, 1, 4, 5, 7, 8, 10, 11, 12)	0.04762	0.03705	0.018525	404	9	0.696996
10	(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)	[0.7769593921905563, 0.6884741223718953, 0.702...	0.710033	(0, 1, 4, 5, 7, 8, 9, 10, 11, 12)	0.046075	0.035848	0.017924	404	10	0.702654
11	(0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12)	[0.754710892556849, 0.6959627893665097, 0.7017...	0.708109	(0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12)	0.035367	0.027516	0.013758	404	11	0.699918
12	(0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12)	[0.7532855958710695, 0.6944570477695307, 0.693...	0.70581	(0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12)	0.035641	0.02773	0.013865	404	12	0.696781
13	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	[0.7535027170817178, 0.6920238509138777, 0.682...	0.702512	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)	0.038207	0.029727	0.014863	404	13	0.692596

How to select Backward vs Forward?

- If there are 550 columns
- You want to select 500:
  - Select backward
- If you want best 50:

- Select forward

- If you want best:
  - Select any

Using `sklearn`

- We did this with the help of `mlxtend`
  - **`from mlxtend.feature_selection import SequentialFeatureSelector as SFS`**
- Now, we'll do the same with `sklearn`

```
from sklearn.feature_selection import SequentialFeatureSelector as SFS



sfs2 = SFS(model,
          n_features_to_select=5,
          direction='forward',
          scoring='r2',
          n_jobs=-1,
          cv=5)

sfs2 = sfs2.fit(X_train, y_train)
```

- Use this when you know how many features you want

There's a parameter called `tol` **(tolerance)** to control the threshold for stopping the feature selection process.

```
selected_features = sfs2.get_support(indices=True)
selected_features

Output:
array([ 0,  1,  4,  5,  7,  8,  9, 10, 11, 12], dtype=int64)
```

- Transform data:

```
X_train_sfs = sfs2.transform(X_train)
X_test_sfs = sfs2.transform(X_test)
```

```
model.fit(X_train_sfs, y_train)

# Evaluate performance
train_score = model.score(X_train_sfs, y_train)
test_score = model.score(X_test_sfs, y_test)
print(f"Train R²: {train_score:.3f}, Test R²: {test_score:.3f}")

Output:
Train R²: 0.725, Test R²: 0.755
```

Purpose of `tol`:

- It determines the minimum improvement in the performance metric (e.g., accuracy, R^2 score) required to continue adding or removing features.
- If the performance doesn't improve by more than the specified `tol` value, the algorithm stops the feature selection process early.

Example:

- `tol=0.01` means that if the improvement in the performance metric is less than 0.01, the feature selection process will halt.

In short, `tol` helps avoid unnecessary feature selection steps when performance improvements become marginal.

Advantages of wrapper Methods

- Accuracy
- Interaction of Features

Disadvantages of wrapper Methods

- Computational Complexity
- Risk of Overfitting
- Model Specific

Recursive Feature Elimination (RFE)

- First it takes all the features and then eliminates features 1 by 1 or in groups.
- It's done by the model you've provided.
- And it does this until the number of features you have provided is reached.

Core Idea:

- **Start** with **all the features**.
- Train a model (such as linear regression, logistic regression, or any estimator that provides some measure of feature importance).
- Evaluate the importance of each feature.
- Remove the least important feature(s).
- Refit the model on the reduced set of features.
- Repeat the process until a desired number of features is reached.

RFE Variant:

RFECV (Recursive Feature Elimination with Cross-Validation): This is a variation of RFE that uses **cross-validation** to find the optimal number of features. It helps to prevent overfitting and select the best subset of features.

```
from sklearn.feature_selection import RFECV

rfecv = RFECV(estimator=model, step=1, cv=5)
rfecv.fit(X_train, y_train)
```

Outputs

After fitting RFE:

- `support_`: Boolean mask indicating selected features (e.g., `[True, False, True]`).
- `ranking_`: Numerical ranking of features (e.g., `[3, 1, 2]`), where `1` = most important.
- `n_features_`: Number of selected features.

Python code:

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression

# Sample data: X (features), y (target)
model = LinearRegression()

# Initialize RFE to select 5 features, removing 1 per iteration
rfe = RFE(
    estimator=model,
    n_features_to_select=5,
    step=1,
    verbose=1
)

# Fit RFE to the data
rfe.fit(X, y)
```

```
# Get selected features
selected_features = X.columns[rfe.support_]
print("Selected features:", selected_features)

# Feature rankings
print("Feature rankings:", rfe.ranking_)
```

Output:

Fitting estimator with 13 features.
Fitting estimator with 12 features.
Fitting estimator with 11 features.
Fitting estimator with 10 features.
Fitting estimator with 9 features.
Fitting estimator with 8 features.
Fitting estimator with 7 features.
Fitting estimator with 6 features.
Selected features: Index(['chas', 'nox', 'rm', 'dis', 'ptratio'], dtype='object')
Feature rankings: [4 6 5 1 1 1 9 1 3 7 1 8 2]

`verbose` → Controls output verbosity (e.g., `0` for silent, `1` for progress updates).

RFE with Cross-Validation (RFECV)

```
from sklearn.feature_selection import RFECV

# Initialize RFECV with cross-validation
rfecv = RFECV(
    estimator=model,
    step=1,
    cv=5,  # 5-fold cross-validation
    scoring="r2"
)

rfecv.fit(X, y)
```

```
# Optimal number of features
print("Optimal features:", rfecv.n_features_)

Output:
Optimal features: 6
```

`step=1` → one feature is removed each time

Advantages

- **Model-Specific Optimization**: Tailors feature selection to the underlying model.
- **Dynamic Elimination**: Adjusts feature sets iteratively based on model feedback.
- **Flexibility**: Works with any model providing feature importance metrics.

Limitations

- **Computational Cost**: Repeated model training can be slow for large datasets.
- **Model Bias**: Feature selection depends on the estimator's accuracy (e.g., poor model = poor selection).
- **Overfitting Risk**: Without cross-validation, the selected features may not generalize.

Best Practices

1. **Use Cross-Validation**: Prefer `RFECV` to automatically determine the best feature count.
2. **Choose Appropriate Models**: Ensure the estimator reliably ranks features (e.g., avoid non-linear models without clear importance metrics).
3. **Scale Features**: Normalize/standardize data for models sensitive to feature scales (e.g., SVMs, linear regression).

4. **Validate Performance**: Test the final model on a holdout set to ensure robustness.

Applications:

- **High-Dimensional Data**: Gene expression analysis, text classification.
- **Model Simplification**: Reduce complexity in deployed models.
- **Interpretability**: Identify key drivers in business analytics.

11. Comparison to Other Methods

Method	Type	Pros	Cons
RFE	Wrapper	Model-specific, dynamic	Computationally expensive
Filter Methods	Stat-based	Fast, model-agnostic	Ignores feature interactions
Embedded	Model-inherent	Efficient (e.g., Lasso, decision trees)	Limited to specific models

RFECV vs. RFE:

RFE (Recursive Feature Elimination):

- RFE performs **recursive feature elimination** by repeatedly fitting the model and eliminating the least important features based on feature importance (coefficients or Gini importance).
- It continues until a predefined number of features is selected.
- **No cross-validation is used** in RFE. The user specifies how many features they want to keep, and RFE removes features step by step.

RFECV (Recursive Feature Elimination with Cross-Validation):

- RFECV is similar to RFE but with an added twist: it uses **cross-validation** to evaluate model performance at each step.

- It performs **recursive feature elimination** and at each iteration, it checks the model's performance using cross-validation to determine the optimal number of features.
- **RFECV automatically selects the optimal number of features** by evaluating the performance of the model on each subset of features using cross-validation.

Feature	RFE	RFECV
Cross-validation	Not used	Used to evaluate model performance
Feature selection	User specifies number of features to select	Automatically determines the optimal number of features
Risk of Overfitting	Can overfit without cross-validation	Less prone to overfitting due to cross-validation
Speed	Faster (no cross-validation)	Slower (cross-validation adds overhead)
Suitability	Good when you know the number of features to keep	Better for selecting the optimal number of features automatically
Computational Cost	Lower	Higher due to cross-validation