Building Good Training Sets - Data Preprocessing

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

Outline

- Dealing with Missing Data
- Handling with Categorical Data
- Partitioning a Dataset into a Separate Training and Test Sets
- Feature Scaling: Bring Different Features onto the Same Scale
- Feature Selection: Selecting Meaningful Features

Dealing with Missing Data

Identifying Missing Values in Tabular Data

```
import pandas as pd
from io import StringIO
import sys

csv_data = \
'''A,B,C,D
1.0,2.0,3.0,4.0
5.0,6.0,,8.0
10.0,11.0,12.0,'''

# If you are using Python 2.7, you need
# to convert the string to unicode:

if (sys.version_info < (3, 0)):
    csv_data = unicode(csv_data)

df = pd.read_csv(StringIO(csv_data))
df
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

```
In [3]: df.isnull()

Out[3]:        A      B      C      D
        0  False  False  False  False
        1  False  False   True  False
        2  False  False  False   True

In [4]: df.isnull().sum()

Out[4]: A    0
        B    0
        C    1
        D    1
        dtype: int64

In [5]: # access the underlying NumPy array
        # via the `values` attribute
        df.values

Out[5]: array([[ 1.,  2.,  3.,  4.],
               [ 5.,  6., nan,  8.],
               [10., 11., 12., nan]])
```

Eliminating Instances or Features with Missing Values

- The easiest way to deal with missing data is to remove. Use **dropna** method

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

```
# remove rows that contain missing values

df.dropna(axis=0)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

```
# remove columns that contain missing values

df.dropna(axis=1)
```

	A	B
0	1.0	2.0
1	5.0	6.0
2	10.0	11.0

```
# remove columns that contain missing values

df.dropna(axis=1)
```

	A	B
0	1.0	2.0
1	5.0	6.0
2	10.0	11.0

```
# only drop rows where all columns are NaN

df.dropna(how='all')
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

```
# drop rows that have less than 3 real values

df.dropna(thresh=4)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

```
# only drop rows where NaN appear in specific columns (here: 'C')

df.dropna(subset=['C'])
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
2	10.0	11.0	12.0	NaN

Imputing (Interpolating) Missing Values

- Use **Imputer** class for mean imputation
  - Replace missing value with the mean value of the entire feature column

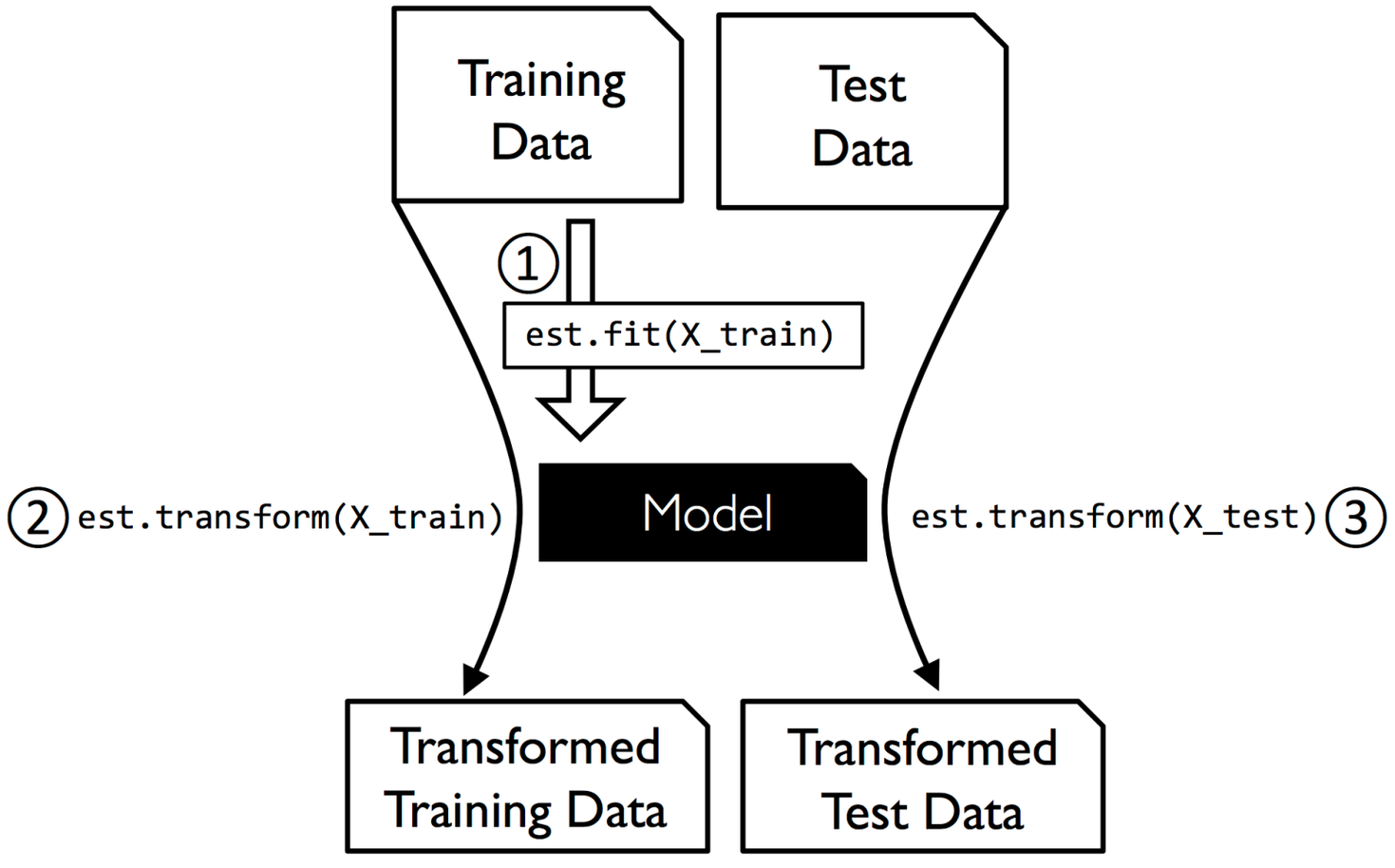
	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

```
# impute missing values via the column mean

from sklearn.preprocessing import Imputer

imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
imputed_data
```

```
array([[  1. ,   2. ,   3. ,   4. ],
       [  5. ,   6. ,   7.5,   8. ],
       [ 10. ,  11. ,  12. ,   6. ]])
```

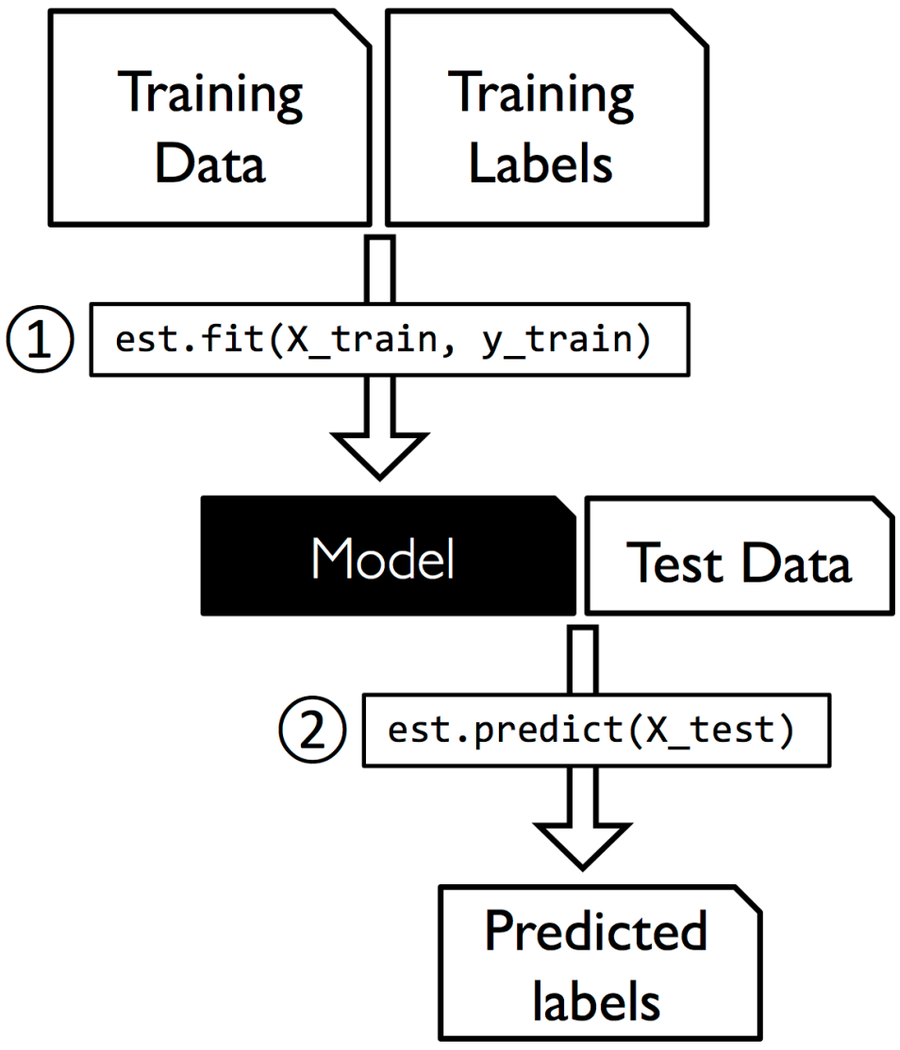
Transformer API in Scikit-learn

- The **imputer** class belongs to **transformer** class in scikit-learn
- The **fit** and **transform** are two essential methods for **transformer** class



Estimator API in Scikit-learn

- Various classifiers in scikit-learn belongs to **estimator** class
- **fit** and **predict** are two essential methods of the **estimator** class
- The **estimator** class also have a **transform** method



Handling Categorical Data

Nominal and Ordinal Features

- Ordinal features can be understood as categorical values that can be sorted or ordered, but nominal features are not.

```
import pandas as pd


df = pd.DataFrame([['green', 'M', 10.1, 'class1'],
                   ['red', 'L', 13.5, 'class2'],
                   ['blue', 'XL', 15.3, 'class1']])

df.columns = ['color', 'size', 'price', 'classlabel']
df
```

	color	size	price	classlabel
0	green	M	10.1	class1
1	red	L	13.5	class2
2	blue	XL	15.3	class1

nominal ordinal numerical

Mapping Ordinal Features

- We usually use ***dictionary mapping*** to map values of an ordinal feature to integers
- We can use ***reverse-dictionary mapping*** to transform the integer values back to the original string values of an ordinal feature

```
size_mapping = {'XL': 3,
                'L': 2,
                'M': 1}

df['size'] = df['size'].map(size_mapping)
df
```

	color	size	price	classlabel
0	green	1	10.1	class1
1	red	2	13.5	class2
2	blue	3	15.3	class1

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
df['size'].map(inv_size_mapping)
```

```
0      M
1      L
2     XL
Name: size, dtype: object
```

Encoding Class Labels (1/3)

- Many ML libraries require that class labels are encoded as integers
- In scikit-learn, most classifiers convert classes to integers internally
- This can be done by creating a mapping dictionary

```
import numpy as np


# create a mapping dict
# to convert class labels from strings to integers
class_mapping = {label: idx for idx, label in enumerate(np.unique(df['classlabel']))}
class_mapping
```

```
{'class1': 0, 'class2': 1}
```

```
# to convert class labels from strings to integers
df['classlabel'] = df['classlabel'].map(class_mapping)
df
```

	color	size	price	classlabel
0	green	1	10.1	0
1	red	2	13.5	1
2	blue	3	15.3	0

Encoding Class Labels (2/3)

- Define a reverse-mapping dictionary can map the converted class labels back to the original string representation

```
# reverse the class label mapping
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
df
```

	color	size	price	classlabel
0	green	1	10.1	class1
1	red	2	13.5	class2
2	blue	3	15.3	class1

Encoding Class Labels (3/3)

- In scikit-learn, the **preprocessing** module has a **LabelEncoder** class which directly implements the conversion

```
from sklearn.preprocessing import LabelEncoder
```

```
# Label encoding with sklearn's LabelEncoder
class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
y
```

```
array([0, 1, 0])
```

```
# reverse mapping
class_le.inverse_transform(y)
```

```
array(['class1', 'class2', 'class1'], dtype=object)
```

Performing One-Hot Encoding on Nominal Features (1/3)

- Performing label conversion to integers directly for nominal features does not make any reasonable sense since the values of a nominal feature ***do not have*** any ***intrinsic order***

```
X = df[['color', 'size', 'price']].values

color_le = LabelEncoder()
X[:, 0] = color_le.fit_transform(X[:, 0])
X
```

```
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

Performing One-Hot Encoding on Nominal Features (2/3)

- Instead, ***one-hot*** encoding is a common technique for this problem
  - To create a new dummy feature for each value possible value in the nominal feature column
  - If there are three possible values ***red***, ***blue***, ***green*** of the color feature which is nominal, we create three new dummy feature ***color_red***, ***color_blue***, ***color_green***, and binary values are assigned to these new dummy values
  - If an instance has blue color value, the values of the three dummy features will be ***color_red***=0, ***color_blue***=1, ***color_green***=0

Performing One-Hot Encoding on Nominal Features (3/3)

```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X).toarray()
```

```
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

```
# return dense array so that we can skip
# the toarray step

ohe = OneHotEncoder(categorical_features=[0], sparse=False)
ohe.fit_transform(X)
```

```
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

```
# one-hot encoding via pandas

pd.get_dummies(df[['price', 'color', 'size']])
```

	price	size	color_blue	color_green	color_red
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0

```
# multicollinearity guard in get_dummies

pd.get_dummies(df[['price', 'color', 'size']], drop_first=True)
```

	price	size	color_green	color_red
0	10.1	1	1	0
1	13.5	2	0	1
2	15.3	3	0	0

```
# multicollinearity guard for the OneHotEncoder

ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X).toarray()[:, 1:]
```

```
array([[  1. ,   0. ,   1. ,  10.1],
       [  0. ,   1. ,   2. ,  13.5],
       [  0. ,   0. ,   3. ,  15.3]])
```

Partition a Dataset into a Separate Training and Test Sets

Wine Dataset

- 178 wine instances with 13 features

```
df_wine = pd.read_csv('https://archive.ics.uci.edu/'
                      'ml/machine-learning-databases/wine/wine.data',
                      header=None)
df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                   'Proline']

print('Class labels', np.unique(df_wine['Class label']))
df_wine.head()
```

```
Class labels [1 2 3]
```

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyan
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1

Wine Dataset

- Extract feature matrix ***X*** and the class label vector ***y***, both as NumPy array, from ***df_wine***
- Split the data into separate training and test datasets (7:3) by scikit-learn's **train_test_split** function from **model_selection** module
- The stratification is specified by ***y***

```
from sklearn.model_selection import train_test_split

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

X_train, X_test, y_train, y_test =\
    train_test_split(X, y,
                     test_size=0.3,
                     random_state=0,
                     stratify=y)
```

Feature Scaling

Bring Different Features onto the Same Scale

Normalization

- Rescale values in a feature column to a range of [0,1]
- A special case of ***min-max scaling*** $x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$
  - $x_{\min}$ and $x_{\max}$ are the minimum and maximum value of the ith feature column
- Useful when we need values in a bounded interval

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

Standardization

- Rescale values in a feature column to take the form of a standard normal distribution
  - Gaussian with *zero mean* and *unit variance*

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

```
from sklearn.preprocessing import StandardScaler

stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

Feature Selection

Selecting Meaningful Features

Select Meaningful Features

- The common reason for overfitting is that our model is too complex for the giving training dataset
- Common solutions to reduce generalization error
  - Collect more training data
  - Choose a simpler model with few parameters
  - Introduce a penalty for complexity via regularization
  - Reduce the dimensionality of the data

L1 and L2 Regularization

- L2 regularization

$$L2: \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

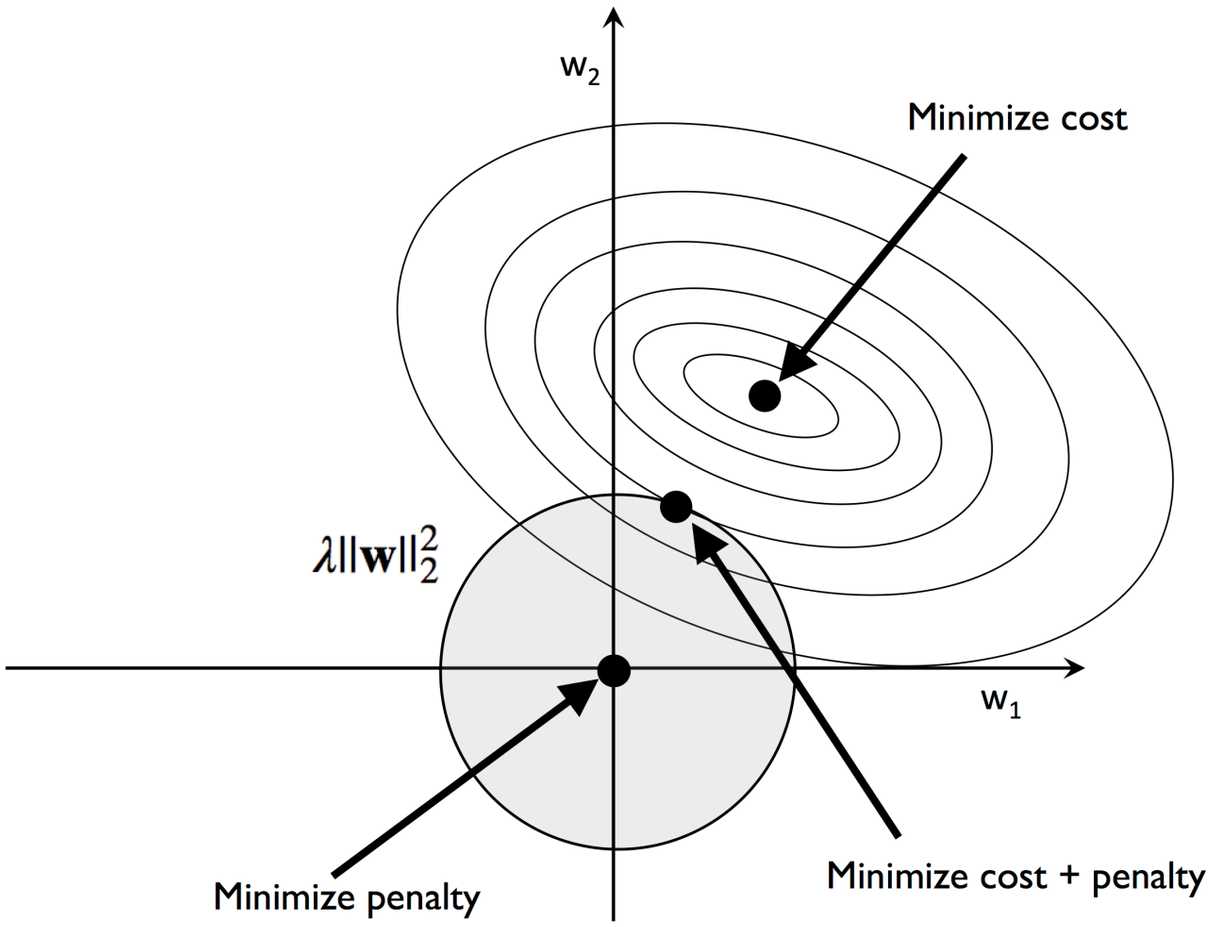
- L1 regularization

$$L1: \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

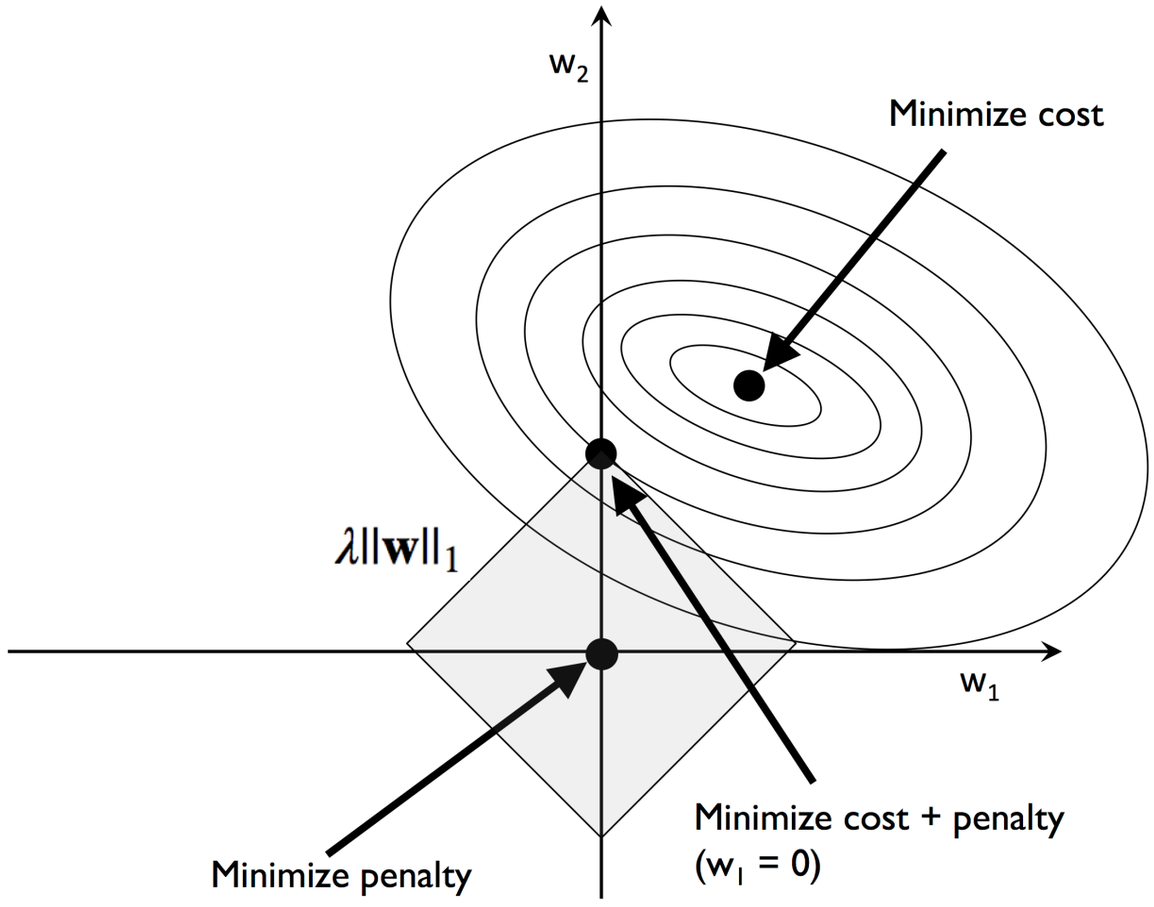
  - L1 yields sparse feature vectors; most feature weights will be zero
  - Useful for high-dimensional datasets with irrelevant features
  - Can be viewed as a technique for feature selection

Geometric Interpretation of L2, L1 Regularization

- Regularization penalty and cost pull in opposite directions
  - Regularization wants the weights to be at (0,0), i.e., prefers a simpler model, and decreases the dependence of the model on training data



L2 Regularization



L1 Regularization

Scikit-lean with L1 Regularization

- For regularized models in scikit-learn that supports L1 regularization, we can set the ***penalty*** parameter to ***l1*** to obtain a sparse solution

```
from sklearn.linear_model import LogisticRegression
LogisticRegression(penalty='l1')
```

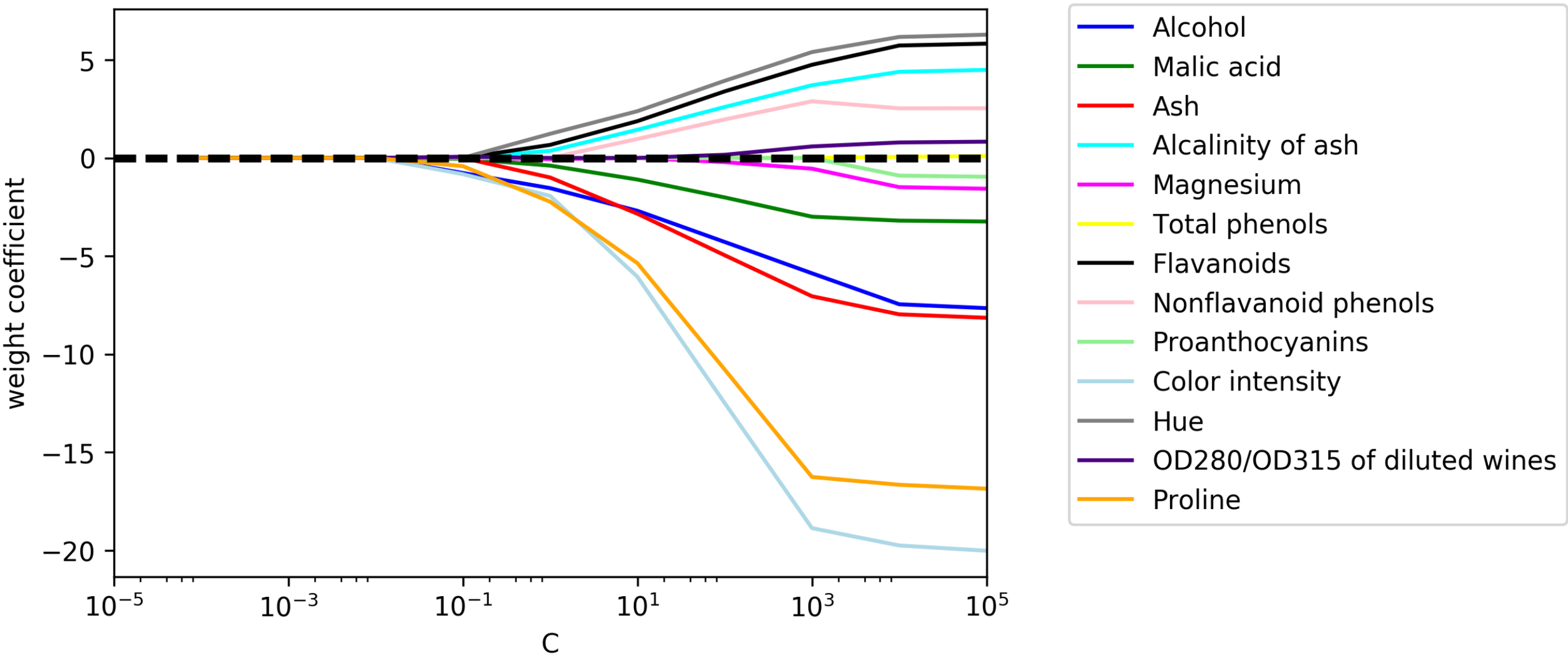
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l1', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

- Apply to the standardized Wine data

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='l1', C=1.0)
lr.fit(X_train_std, y_train)
print('Training accuracy:', lr.score(X_train_std, y_train))
print('Test accuracy:', lr.score(X_test_std, y_test))
```

```
Training accuracy: 1.0
Test accuracy: 1.0
```

L1 Regularization on Wine Data



Dimensionality Reduction of Data

- Two main categories of dimensionality reduction
  - Feature selection
    - Selecting a subset of important features from the original features
  - Feature extraction
    - Deriving new features from the original features by creating a mapping from the original feature space $\mathbb{R}^m$ to a feature space $\mathbb{R}^d$ of lower dimension

Sequential Feature Selection Algorithms

- A family of greedy search algorithms that are used to eliminate ***unimportant*** features from the original features
- Dimensionality reduction of data aims to improve computational efficiency and reduce the generalization error of model by removing irrelevant features or noise
- Useful for learning algorithms which do not support regularization

Sequential Backward Selection (SBS) Algorithm

- Implementation of SBS
  - Initialize the algorithm with $k=d$, where d is the dimensionality of the full feature space $\mathbf{X}_d$
  - Determine the feature x^- that maximizes the criterion: x^- =argmax $J(\mathbf{X}_k - x)$, where $\boldsymbol{x} \in \boldsymbol{X}_k$
  - Remove the feature x^- from the feature set $\mathbf{X}_{k-1}=\mathbf{X}_k - x^-$; $k=k-1$
  - Terminate if k equals the number of desired features; otherwise, go to step 2.

Sequential Feature Selection (SBS) (1/3)

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split


class SBS():
    def __init__(self, estimator, k_features, scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state
```

Sequential Feature Selection (SBS) (2/3)

```
    def fit(self, X, y):

        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self
```

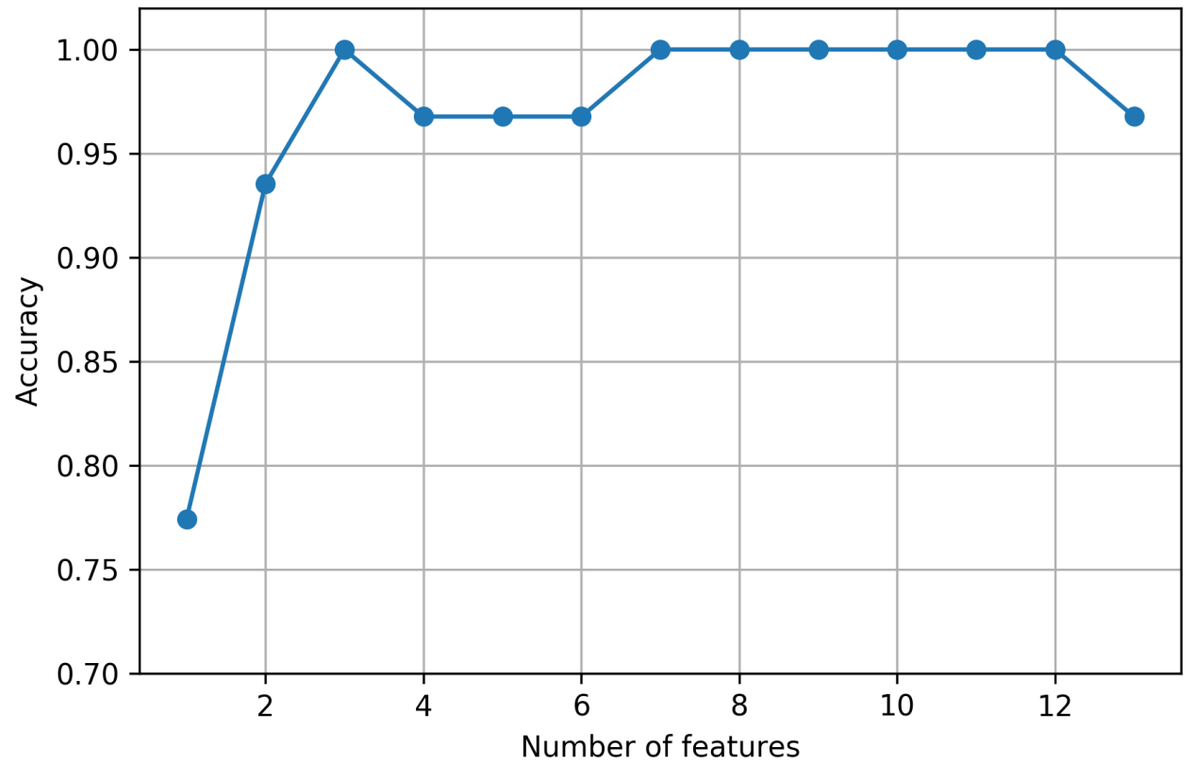
Sequential Feature Selection (SBS) (3/3)

```
    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

KNN Classifier with SBS

```
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)

# selecting features
sbs = SBS(knn, k_features=1)
sbs.fit(X_train_std, y_train)
```

```
# plotting performance of feature subsets
k_feat = [len(k) for k in sbs.subsets_]

plt.plot(k_feat, sbs.scores_, marker='o')
plt.ylim([0.7, 1.02])
plt.ylabel('Accuracy')
plt.xlabel('Number of features')
plt.grid()
plt.tight_layout()
# plt.savefig('images/04_08.png', dpi=300)
plt.show()
```



Assessing Feature Importance with Random Forest

- Measure
  - With K binary decision trees in a random forest, the mean decrease impurity of the ith feature x_i for the random forest is the average of the mean decrease impurity of the K trees
  - The larger the mean decrease impurity, the more important a feature is.

scikit-learn Example

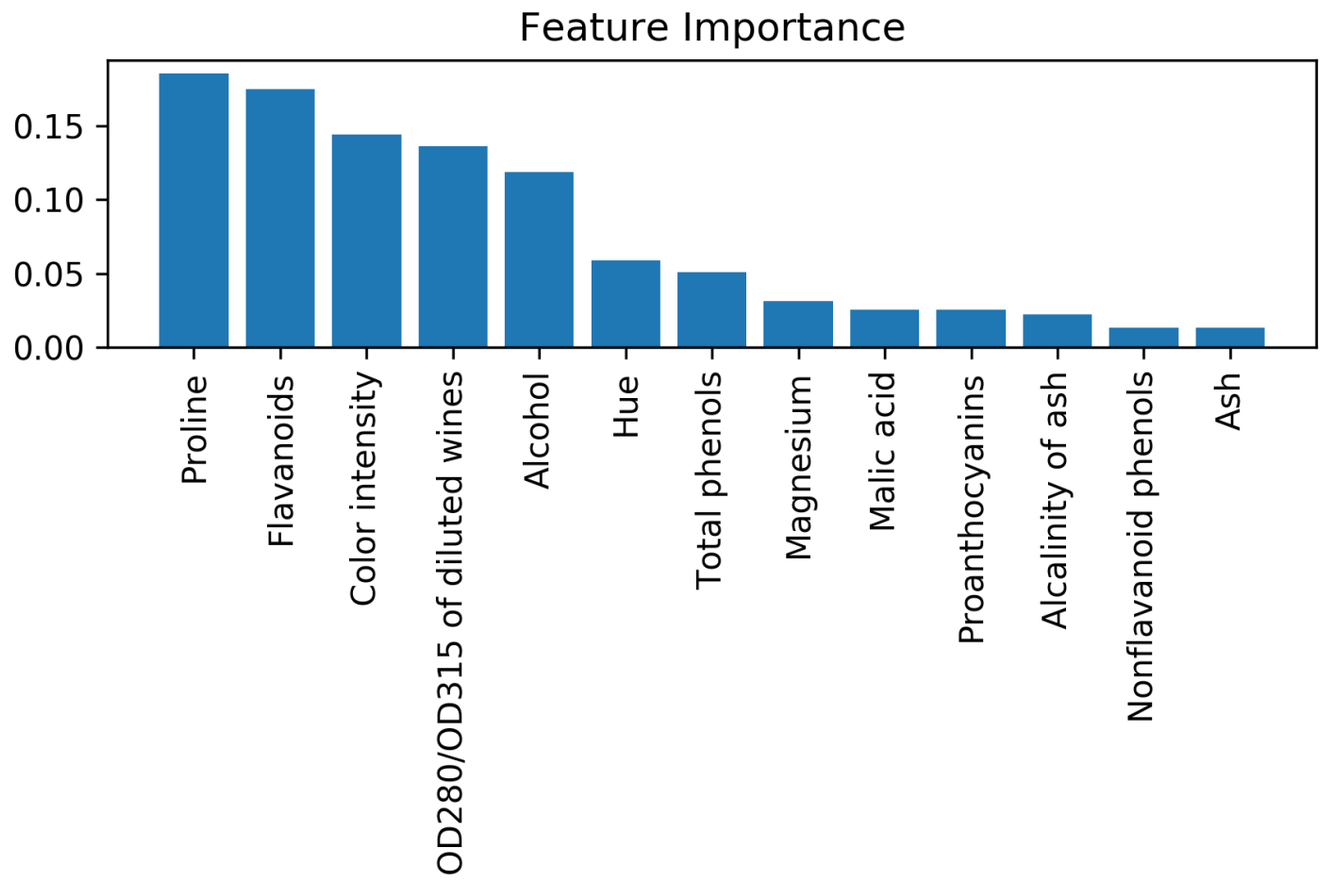
- The **DecisionTreeClassifier** and **RandomForestClassifier** classes in scikit-learn's tree and ensemble modules respectively automatically compute the feature importance and store the values in the *feature_importance_* attribute (which is an array)

```
from sklearn.ensemble import RandomForestClassifier

feat_labels = df_wine.columns[1:]

forest = RandomForestClassifier(n_estimators=500,
                                random_state=1)

forest.fit(X_train, y_train)
importances = forest.feature_importances_

indices = np.argsort(importances)[::-1]
```

```
for f in range(X_train.shape[1]):
    print("%2d) %-*s %f" % (f + 1, 30,
                            feat_labels[indices[f]],
                            importances[indices[f]]))

plt.title('Feature Importance')
plt.bar(range(X_train.shape[1]),
        importances[indices],
        align='center')

plt.xticks(range(X_train.shape[1]),
           feat_labels[indices], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
#plt.savefig('images/04_09.png', dpi=300)
plt.show()
```

Results

```
 1) Proline                        0.185453
 2) Flavanoids                     0.174751
 3) Color intensity                0.143920
 4) OD280/OD315 of diluted wines   0.136162
 5) Alcohol                        0.118529
 6) Hue                            0.058739
 7) Total phenols                  0.050872
 8) Magnesium                      0.031357
 9) Malic acid                     0.025648
10) Proanthocyanins                0.025570
11) Alcalinity of ash              0.022366
12) Nonflavanoid phenols           0.013354
13) Ash                            0.013279
```



scikit-learn Example

- Scikit-learn also implements a **SelectFromModel** class that selects features based on user-specific threshold after model fitting

```
from sklearn.feature_selection import SelectFromModel


sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
X_selected = sfm.transform(X_train)
print('Number of samples that meet this criterion:',
      X_selected.shape[0])
```

```
Number of samples that meet this criterion: 124
```

```
for f in range(X_selected.shape[1]):
    print("%2d) %-*s %f" % (f + 1, 30,
                            feat_labels[indices[f]],
                            importances[indices[f]]))
```

```
 1) Proline                        0.185453
 2) Flavanoids                     0.174751
 3) Color intensity                0.143920
 4) OD280/OD315 of diluted wines   0.136162
 5) Alcohol                        0.118529
```