Machine Learning with Python – SciKit-learn Part 2

Ramesh Shankar

UConn

Logistic regression

- Limitations of perceptron
  - Easy introduction to machine learning for classification, but
  - Won’t converge if categories not perfectly linearly separable
    - E.g. weights could be updated indefinitely if even one misclassification
  - Another alternative: Logistic Regression
- Logistic regression
  - Classification technique
  - Works well for linearly separable categories
  - One of most widely used classification algorithms in industry
  - Linear model for binary classification (like Perceptron)
  - Can also be used for multi-class classification

Logistic regression

- Probability of event we want to predict: p
- Odds ratio: p/(1-p)
- Logit(p) = log(p/(1-p)) ➔ inverse of logit function is the logistic function ϕ(z)

$\phi(z) = p$: probability of our event of interest

$$where\ z = \boldsymbol{w}^T.\boldsymbol{x} = \boldsymbol{logit}(\boldsymbol{p}) = \boldsymbol{log}\frac{\boldsymbol{p}}{\boldsymbol{1-p}}$$

where $\boldsymbol{w} = \begin{bmatrix} w_1 \\ \cdots \\ w_m \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x_1 \\ \cdots \\ x_m \end{bmatrix}$

$$z = \boldsymbol{log}\frac{\boldsymbol{p}}{\boldsymbol{1-p}}$$

Implies

$$p = \frac{\boldsymbol{1}}{\boldsymbol{1+e^{-z}}}$$

i.e.

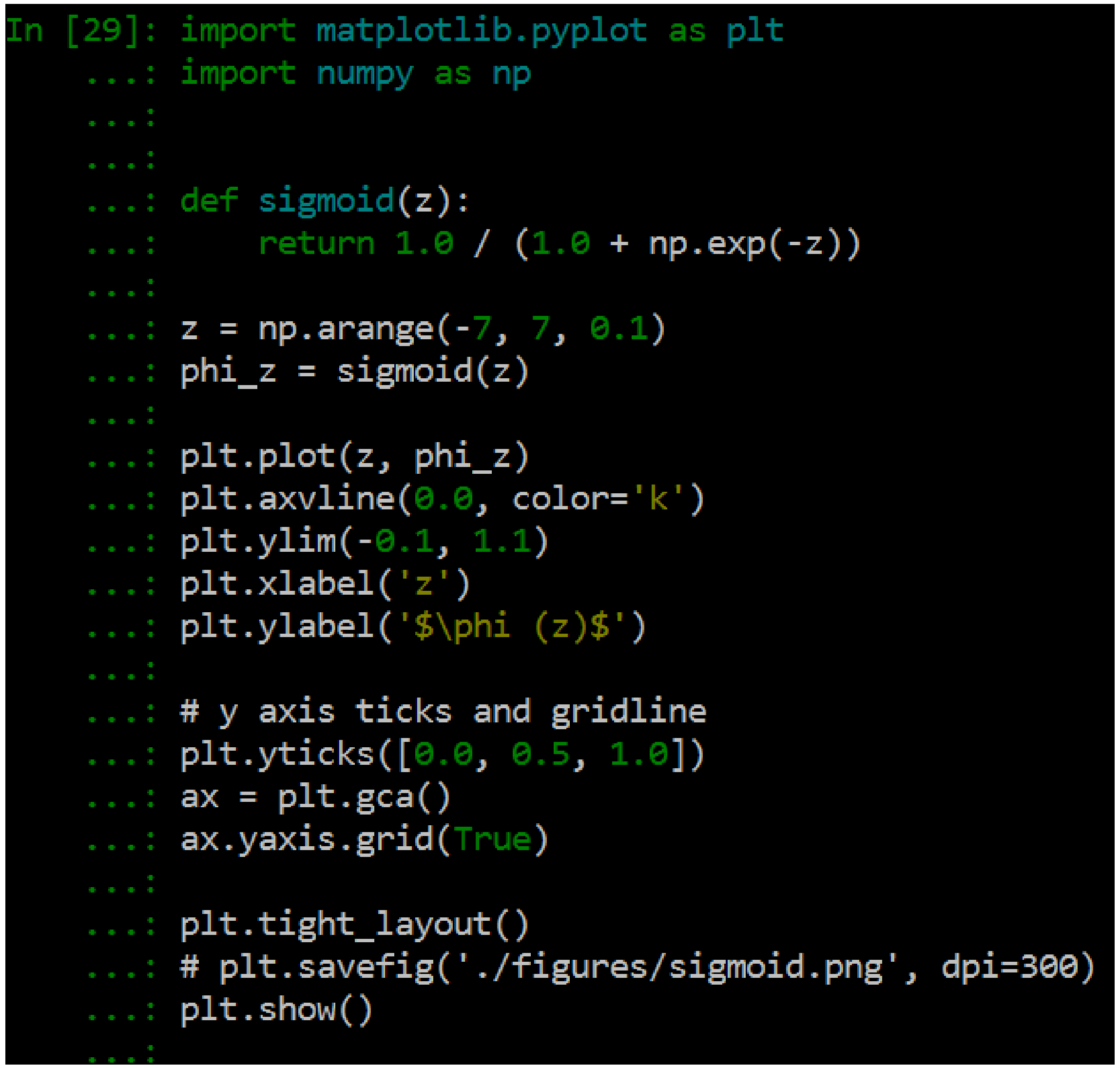
$$\phi(z) = \frac{\boldsymbol{1}}{\boldsymbol{1+e^{-z}}}$$

As $z \rightarrow \infty, \phi(z) \rightarrow 1$

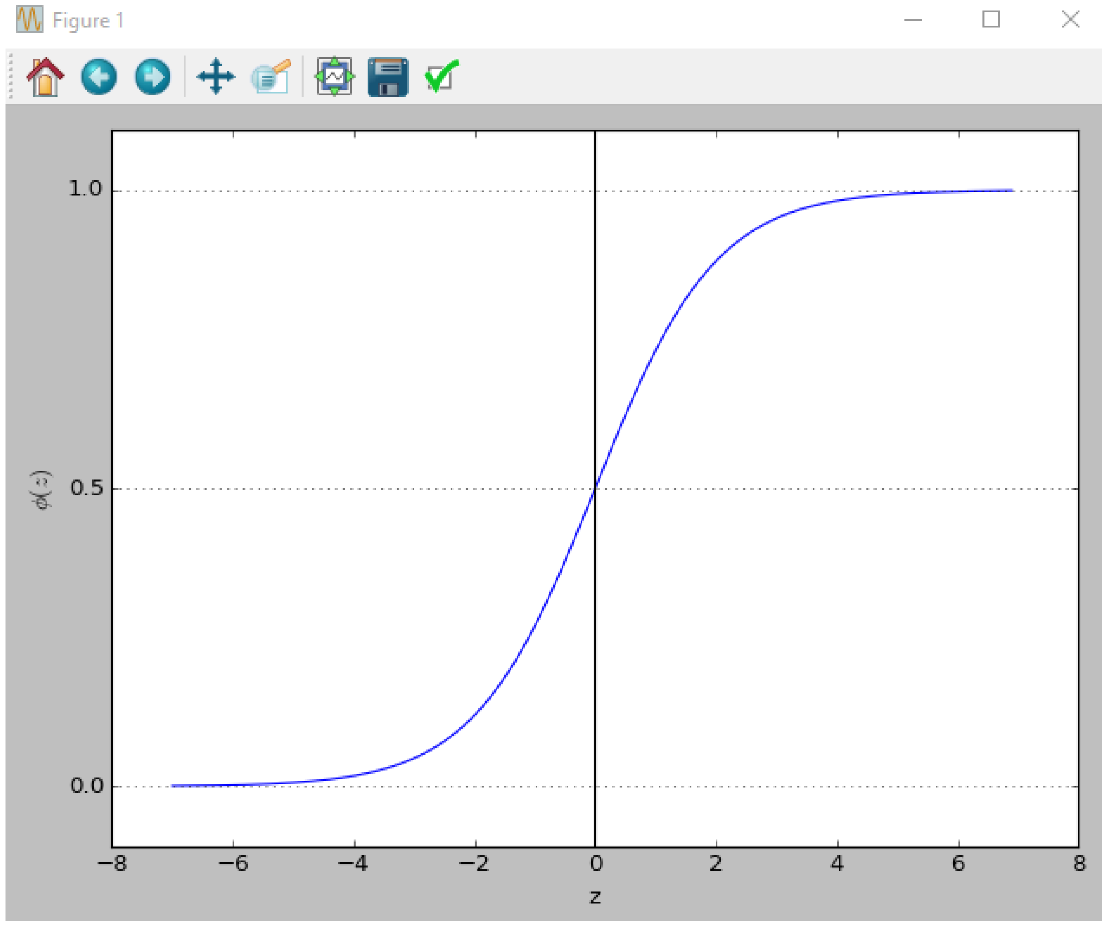
As $z \rightarrow -\infty, \phi(z) \rightarrow 0$

$$\phi(z) = P(y = 1|\boldsymbol{x};\boldsymbol{w})$$

What does the sigmoid curve look like?



```
In [29]: import matplotlib.pyplot as plt
    ...: import numpy as np
    ...:
    ...:
    ...: def sigmoid(z):
    ...:     return 1.0 / (1.0 + np.exp(-z))
    ...:
    ...: z = np.arange(-7, 7, 0.1)
    ...: phi_z = sigmoid(z)
    ...:
    ...: plt.plot(z, phi_z)
    ...: plt.axvline(0.0, color='k')
    ...: plt.ylim(-0.1, 1.1)
    ...: plt.xlabel('z')
    ...: plt.ylabel('$\phi (z)$')
    ...:
    ...: # y axis ticks and gridline
    ...: plt.yticks([0.0, 0.5, 1.0])
    ...: ax = plt.gca()
    ...: ax.yaxis.grid(True)
    ...:
    ...: plt.tight_layout()
    ...: # plt.savefig('./figures/sigmoid.png', dpi=300)
    ...: plt.show()
    ...:
```



Logistic regression

Predicted probability converted to binary outcome via unit step function:

$$\hat{y} = \begin{cases} 1 & \mathit{if}\, \phi(z) \geq 0.5 \\ 0 & \mathit{otherwise} \end{cases}$$

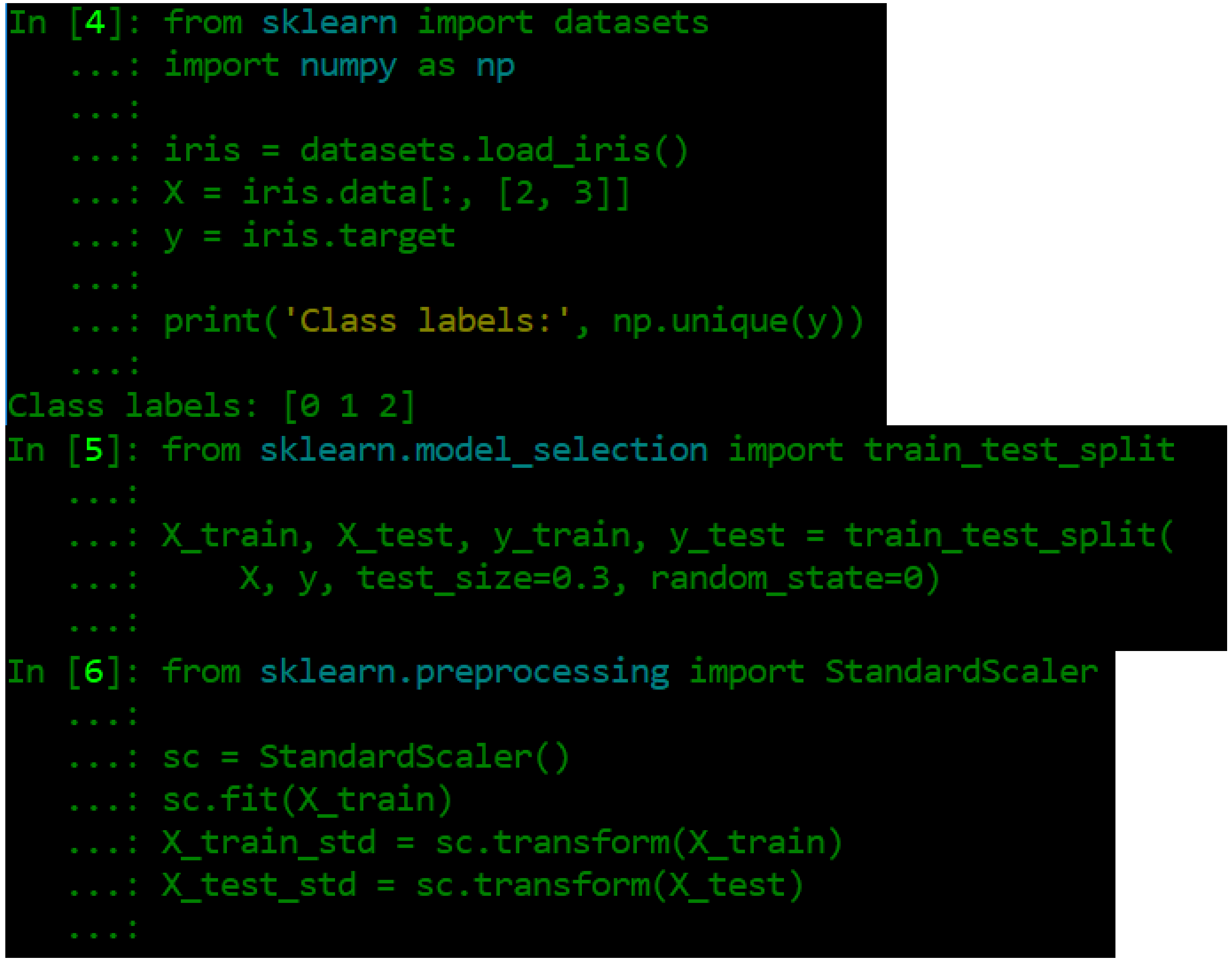
Equivalently:

$$\hat{y} = \begin{cases} 1 & \mathit{if}\ z \geq 0.0 \\ 0 & \mathit{otherwise} \end{cases}$$

Applications of logistic regression

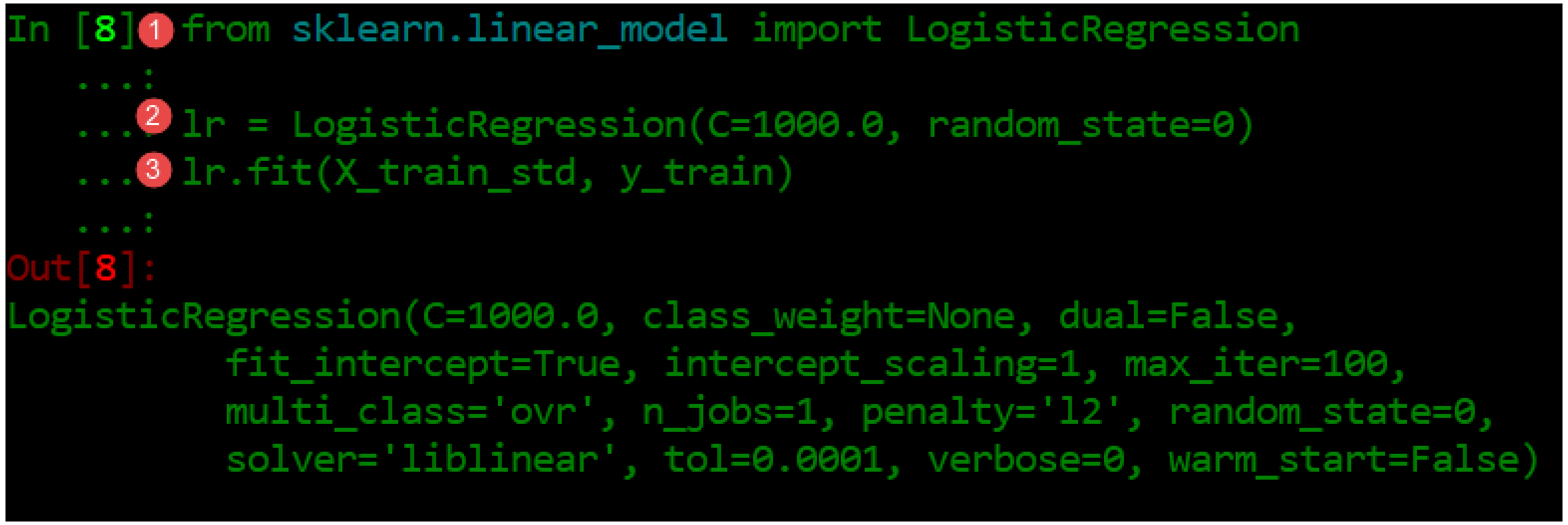
- Weather forecasting – e.g. probability of rain
- Marketing/Sales: Likelihood of a customer purchasing a product
- HR/Management: Likelihood of an employee leaving or joining a firm
- Accounting: Likelihood of an accounting transaction being fraudulent
- Manufacturing/operations: Likeliood of a part being defective
- IT: Likelihood of a web log entry being a hacking attempt

Ingesting and preparing data (repeated)



```
In [4]: from sklearn import datasets
   ...: import numpy as np
   ...:
   ...: iris = datasets.load_iris()
   ...: X = iris.data[:, [2, 3]]
   ...: y = iris.target
   ...:
   ...: print('Class labels:', np.unique(y))
   ...:
Class labels: [0 1 2]
In [5]: from sklearn.model_selection import train_test_split
   ...:
   ...: X_train, X_test, y_train, y_test = train_test_split(
   ...:     X, y, test_size=0.3, random_state=0)
   ...:
In [6]: from sklearn.preprocessing import StandardScaler
   ...:
   ...: sc = StandardScaler()
   ...: sc.fit(X_train)
   ...: X_train_std = sc.transform(X_train)
   ...: X_test_std = sc.transform(X_test)
   ...:
```

Perform logistic regression

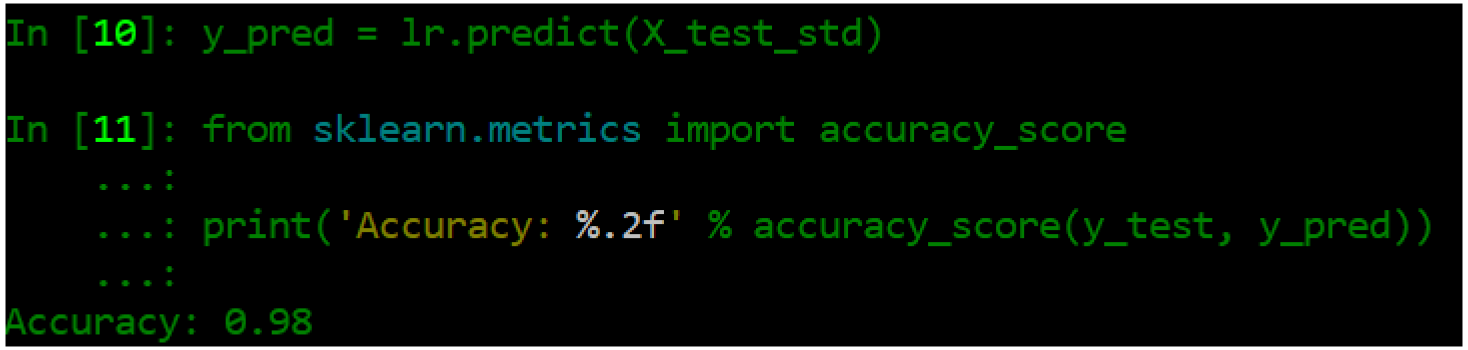


```
In [8]: from sklearn.linear_model import LogisticRegression
   ...:
   ...: lr = LogisticRegression(C=1000.0, random_state=0)
   ...: lr.fit(X_train_std, y_train)
   ...:
Out[8]:
LogisticRegression(C=1000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=0,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

1. Import logistic regression class from scikit-learn
2. Create an object lr of the logistic regression class
    - “C” is a parameter that controls for overfitting – lower C implies less overfitting

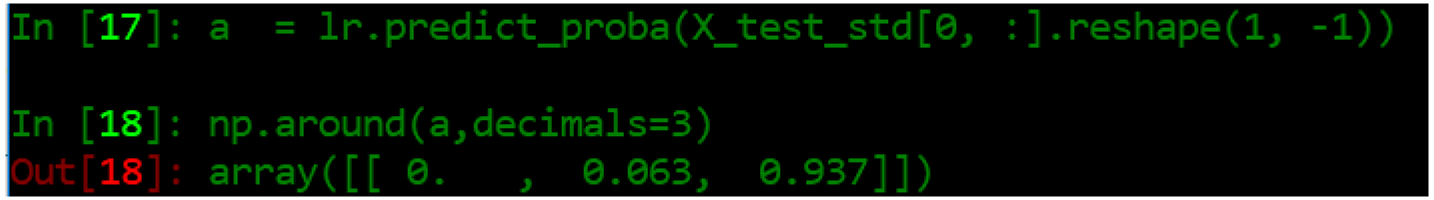
3. Fit the model on the training dataset

Predict using learned model



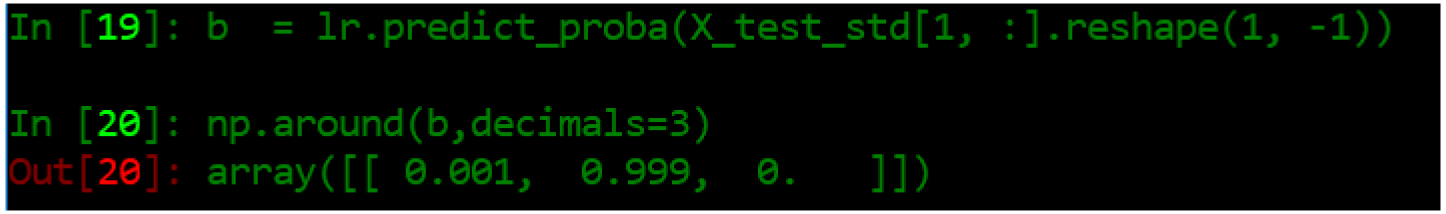
```
In [10]: y_pred = lr.predict(X_test_std)

In [11]: from sklearn.metrics import accuracy_score
    ...:
    ...: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
    ...:
Accuracy: 0.98
```

Predicting individual test sample:

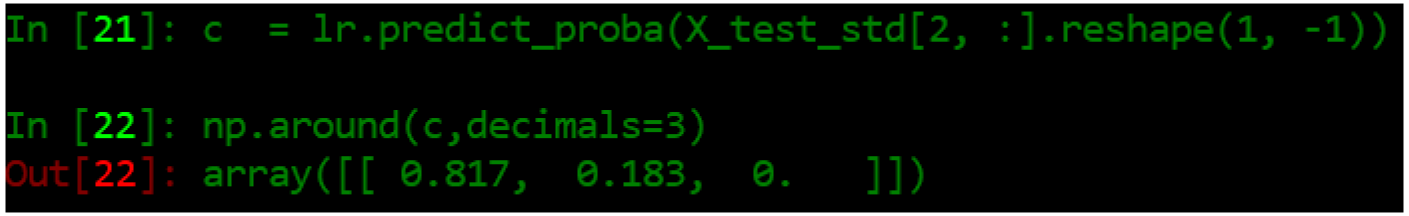


```
In [17]: a  = lr.predict_proba(X_test_std[0, :].reshape(1, -1))

In [18]: np.around(a,decimals=3)
Out[18]: array([[ 0.   ,  0.063,  0.937]])
```

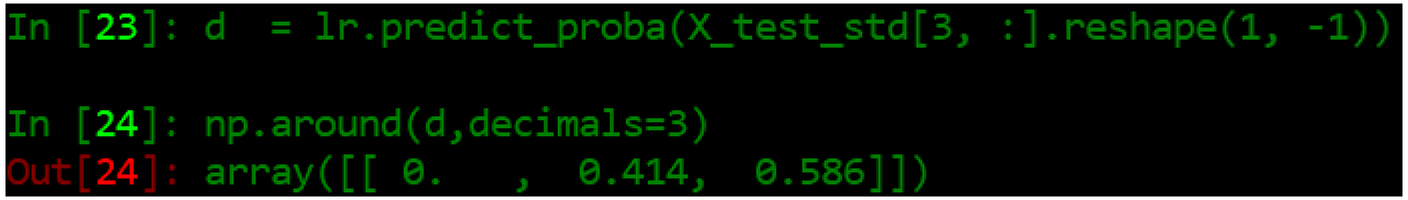
i.e. change of 93.7 percent that sample belongs to Iris-Virginica class,
Chance of 6.3 percent that saple is Iris-Versicolor



```
In [19]: b  = lr.predict_proba(X_test_std[1, :].reshape(1, -1))

In [20]: np.around(b,decimals=3)
Out[20]: array([[ 0.001,  0.999,  0.   ]])
```

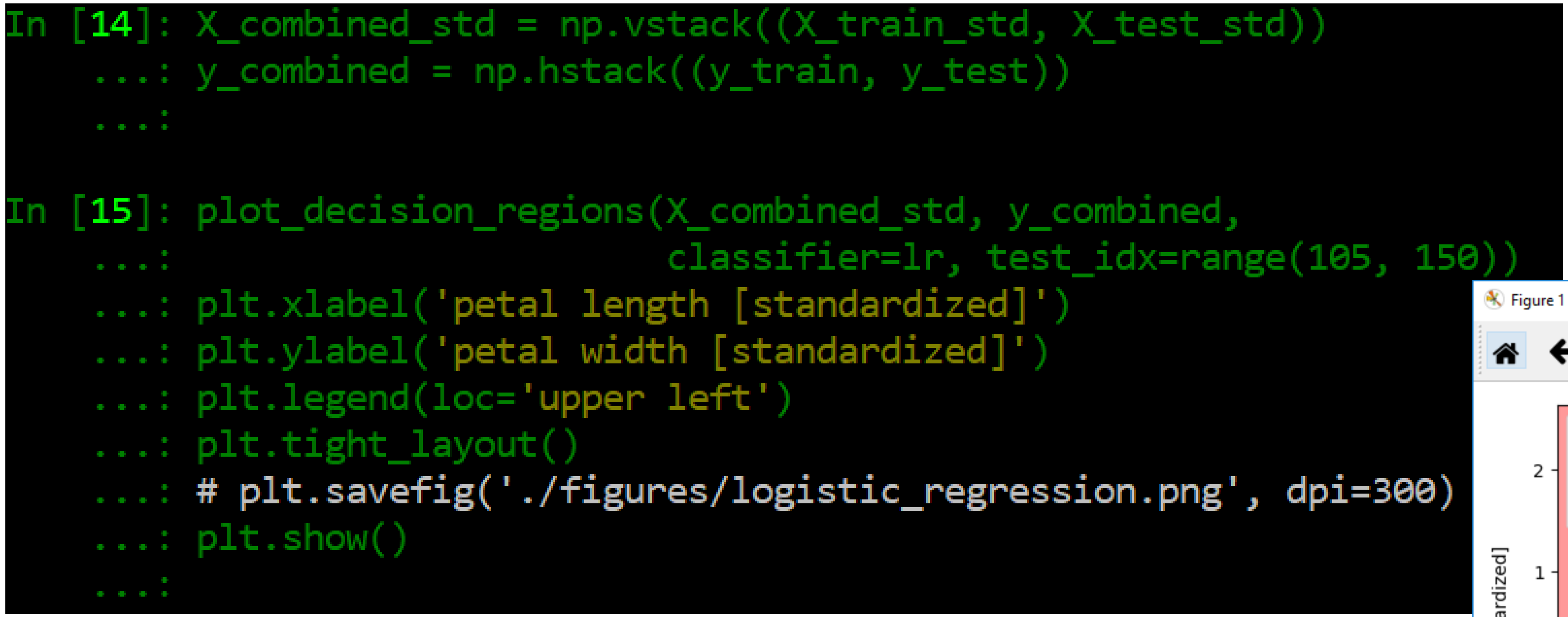


```
In [21]: c  = lr.predict_proba(X_test_std[2, :].reshape(1, -1))

In [22]: np.around(c,decimals=3)
Out[22]: array([[ 0.817,  0.183,  0.   ]])
```

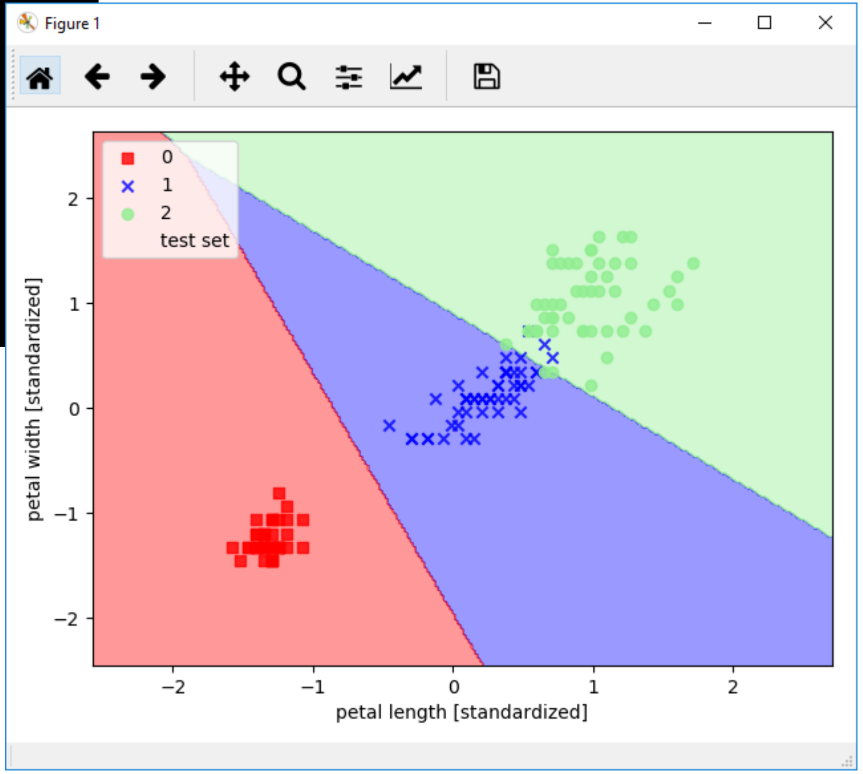


```
In [23]: d  = lr.predict_proba(X_test_std[3, :].reshape(1, -1))

In [24]: np.around(d,decimals=3)
Out[24]: array([[ 0.   ,  0.414,  0.586]])
```

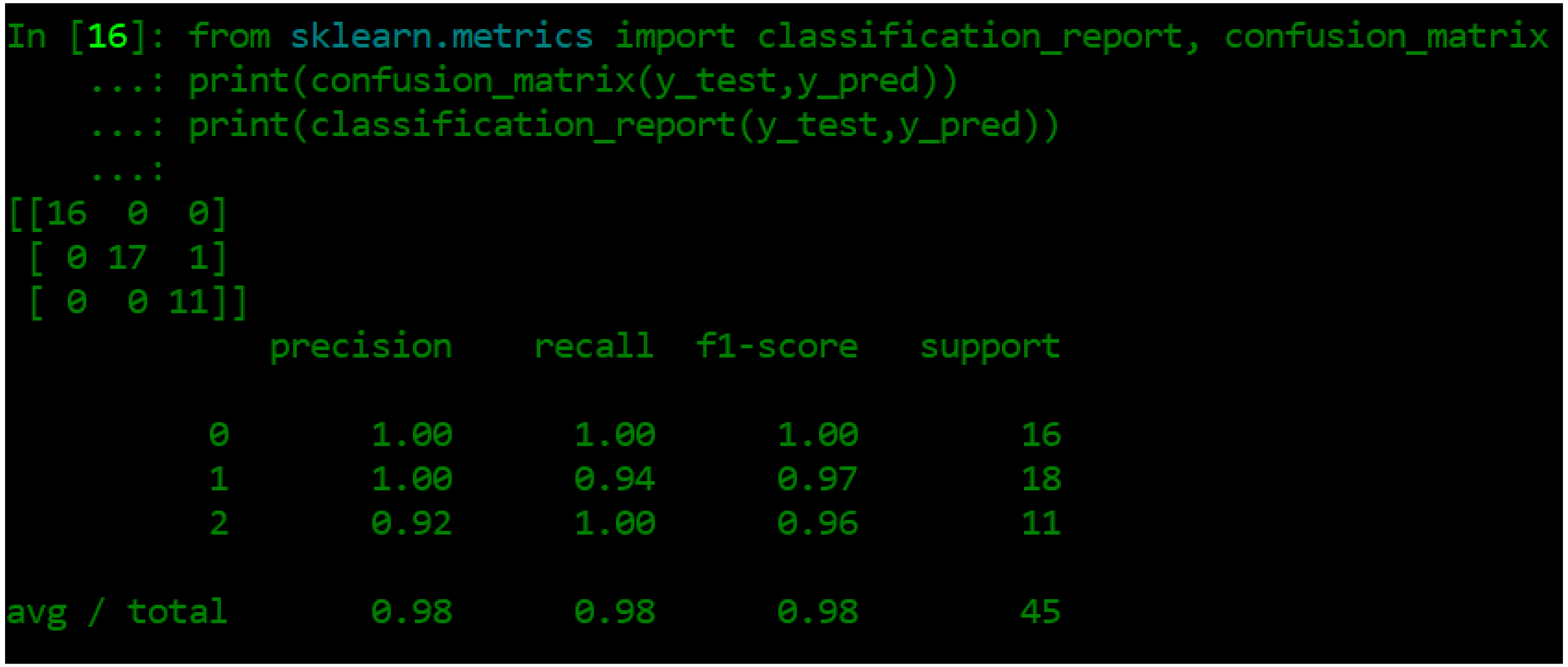
Plot the points



```
In [14]: X_combined_std = np.vstack((X_train_std, X_test_std))
    ...: y_combined = np.hstack((y_train, y_test))
    ...:

In [15]: plot_decision_regions(X_combined_std, y_combined,
    ...:                       classifier=lr, test_idx=range(105, 150))
    ...: plt.xlabel('petal length [standardized]')
    ...: plt.ylabel('petal width [standardized]')
    ...: plt.legend(loc='upper left')
    ...: plt.tight_layout()
    ...: # plt.savefig('./figures/logistic_regression.png', dpi=300)
    ...: plt.show()
    ...:
```

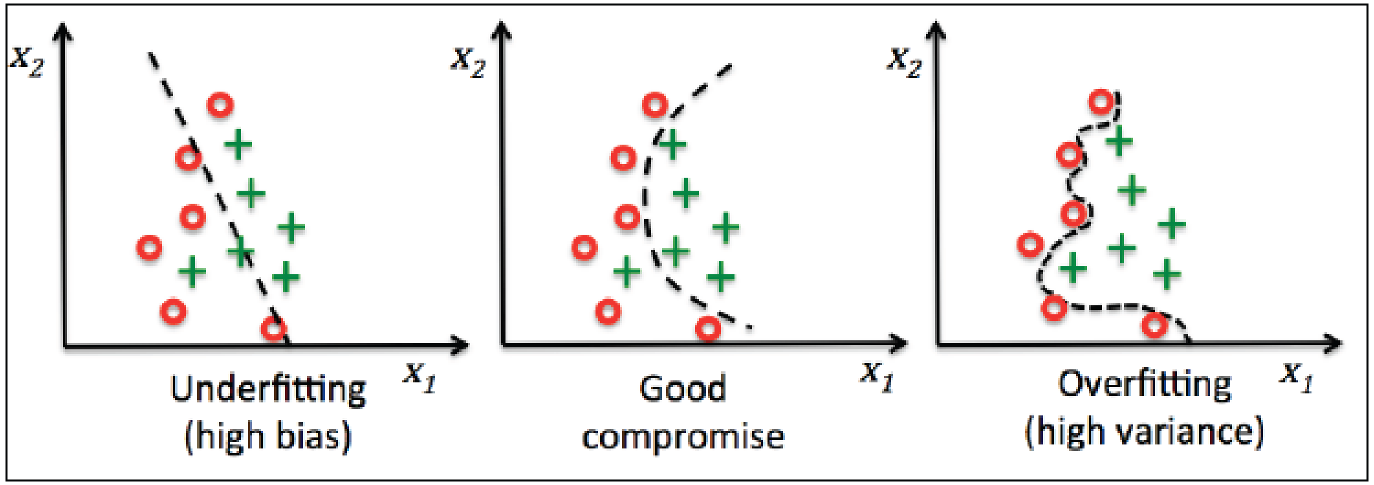


Precision, recall, f1-score



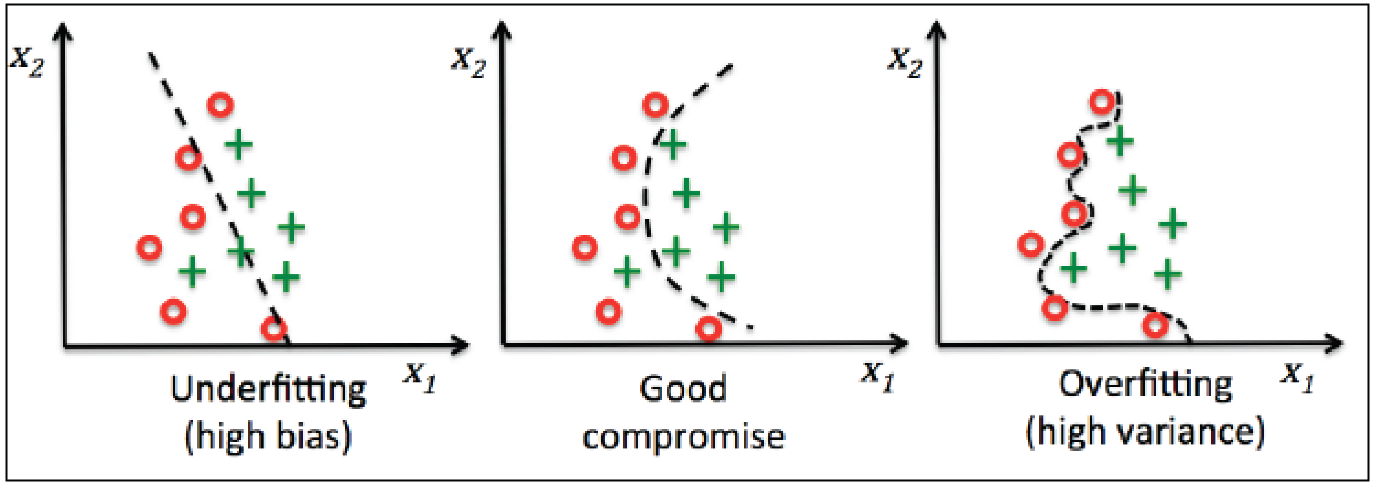
```
In [16]: from sklearn.metrics import classification_report, confusion_matrix
    ...: print(confusion_matrix(y_test,y_pred))
    ...: print(classification_report(y_test,y_pred))
    ...:
[[16  0  0]
 [ 0 17  1]
 [ 0  0 11]]
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        16
          1       1.00      0.94      0.97        18
          2       0.92      1.00      0.96        11

avg / total       0.98      0.98      0.98        45
```

- Overfitting:
  - Model performs well on training data
  - But not on test data
  - Also, we can say model has “high variance”
  - When model is too complex given underlying data
- Underfitting:
  - Not complex enough to capture training data pattern well
  - Therefore, low performance on unseen data

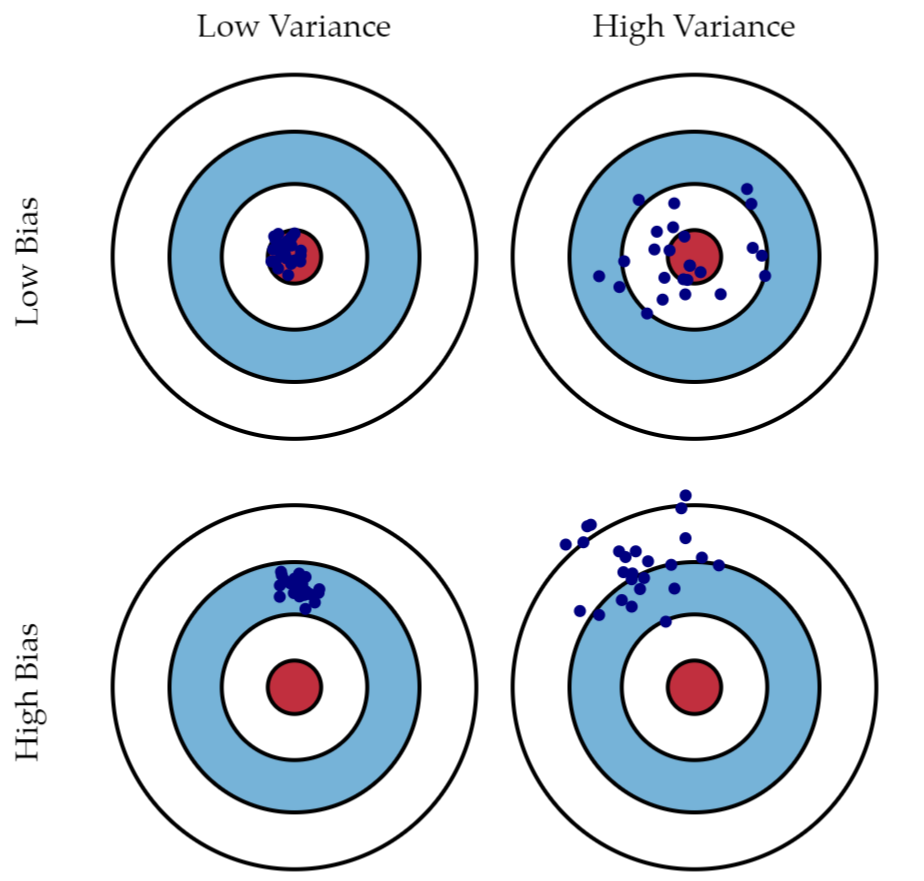


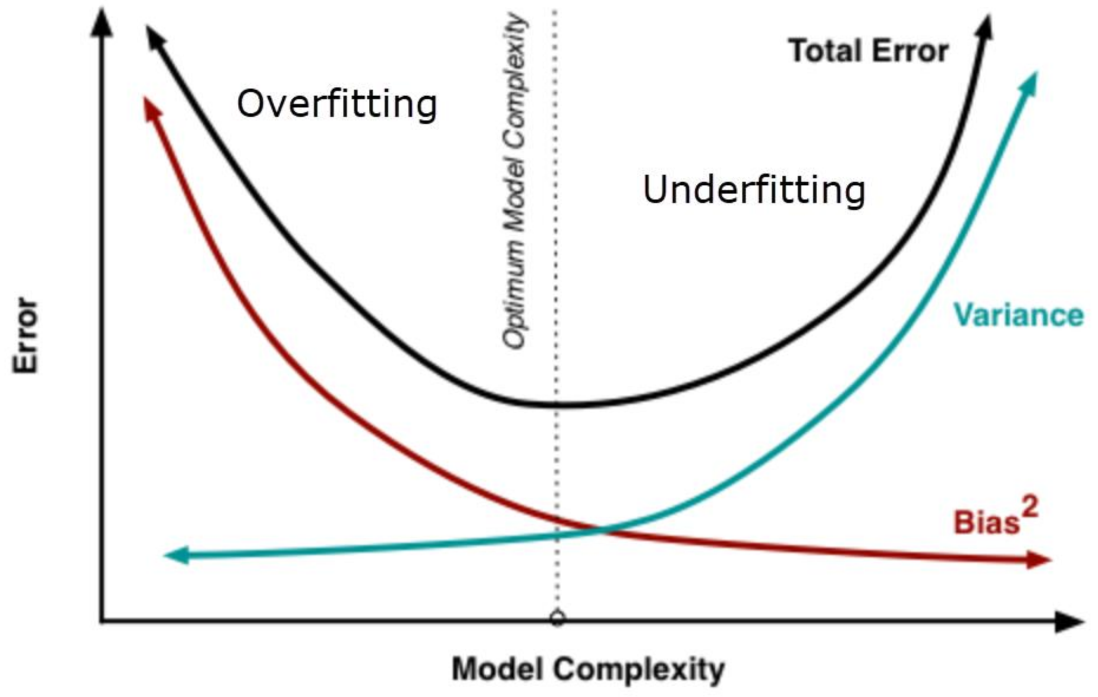
Variance vs Bias

- Variance:
  - Measures variability of model predictions if we retrained model on different subsets of training dataset
  - If variance is high, model is sensitive to randomness in training data
- Bias
  - Measures how far off predictions are from correct values if model is rebuilt multiple times on different training sets
  - Measure of systematic error not due to randomness



Bias and variance





Regularization

- Tune the model complexity via regularization
- Helps find a good bias-variance tradeoff
- Helps to handle
  - high correlation among features (high collinearity)
  - Filter noise from data
  - Prevent overfitting
- Idea:
  - Introduce additional information (bias) – penalize extreme parameter weights
- Most common: L2 regularization (also known as L2 shrinkage or L2 weight decay)

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Regularization

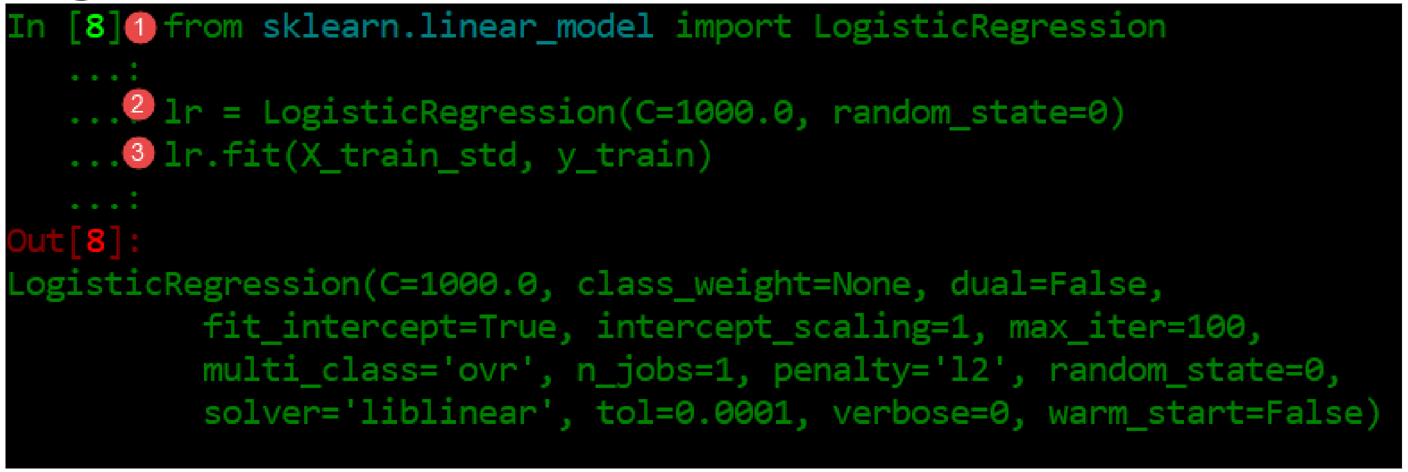
- Add regularization term to cost function used to change weights:

$$J(\boldsymbol{w}) = \left[\sum_{i=1}^{n}\left(-\log\left(\phi\left(z^{(i)}\right)\right) + \left(1-y^{(i)}\right)\left(-\log\left(1-\phi\left(z\right)\right)\right)\right)\right] + \frac{\lambda}{2}\left\|\boldsymbol{w}\right\|^{2}$$

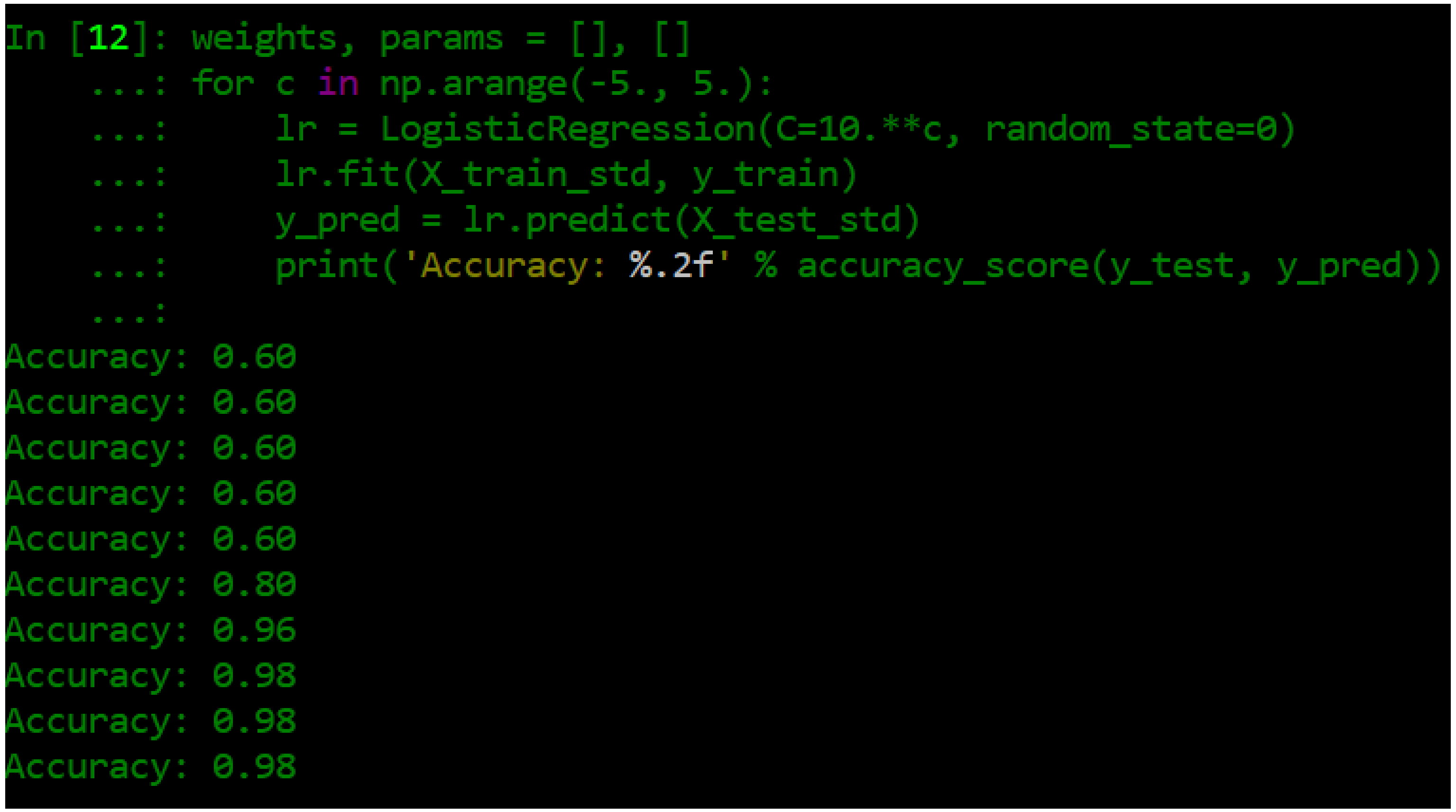
- Parameter “C” in Logistic Regression method:

$$C = \frac{1}{\lambda}$$

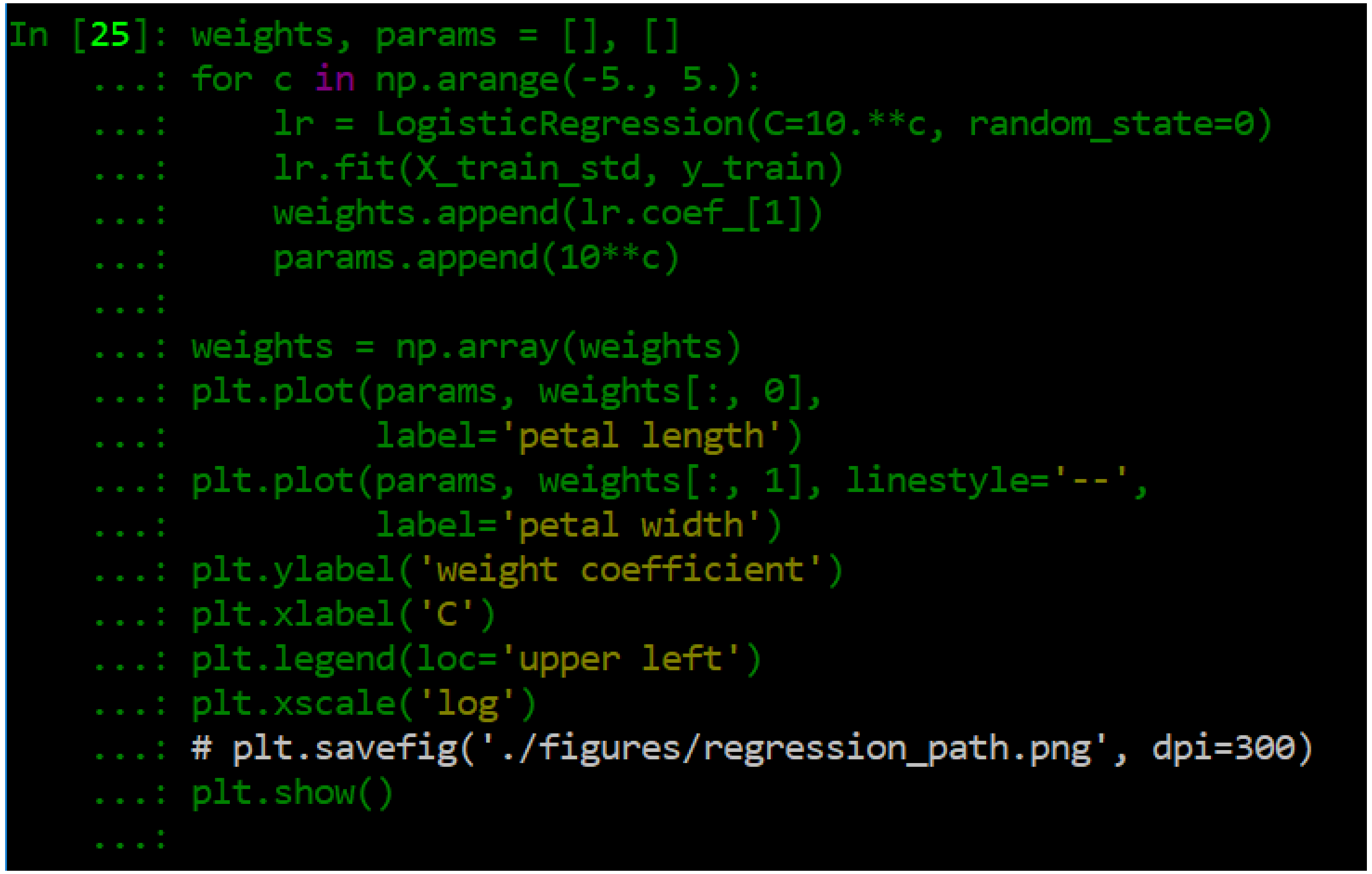
Lower the C, higher the regularization strength – and lesser the overfitting (but more underfitting)



```
In [8]①from sklearn.linear_model import LogisticRegression
   ...:
   ...②lr = LogisticRegression(C=1000.0, random_state=0)
   ...③lr.fit(X_train_std, y_train)
   ...:
Out[8]:
LogisticRegression(C=1000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=0,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

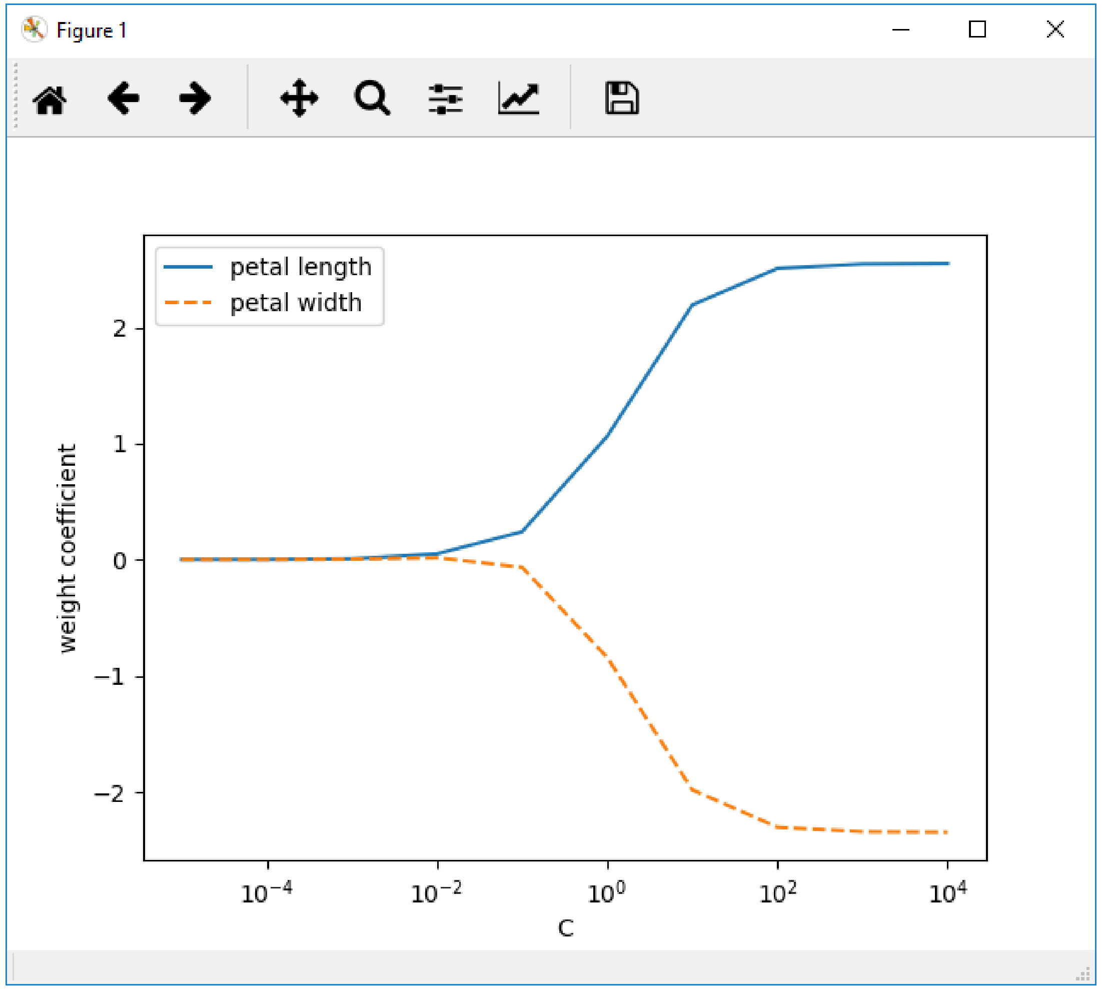


```
In [12]: weights, params = [], []
    ...: for c in np.arange(-5., 5.):
    ...:     lr = LogisticRegression(C=10.**c, random_state=0)
    ...:     lr.fit(X_train_std, y_train)
    ...:     y_pred = lr.predict(X_test_std)
    ...:     print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
    ...:
Accuracy: 0.60
Accuracy: 0.60
Accuracy: 0.60
Accuracy: 0.60
Accuracy: 0.60
Accuracy: 0.80
Accuracy: 0.96
Accuracy: 0.98
Accuracy: 0.98
Accuracy: 0.98
```



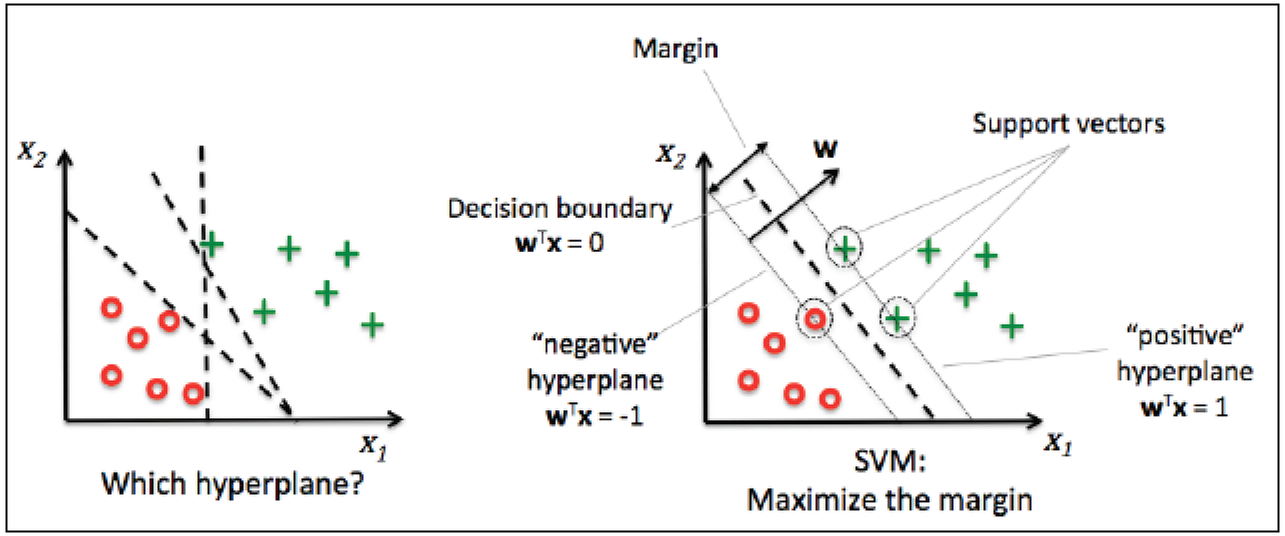
```
In [25]: weights, params = [], []
    ...: for c in np.arange(-5., 5.):
    ...:     lr = LogisticRegression(C=10.**c, random_state=0)
    ...:     lr.fit(X_train_std, y_train)
    ...:     weights.append(lr.coef_[1])
    ...:     params.append(10**c)
    ...:
    ...: weights = np.array(weights)
    ...: plt.plot(params, weights[:, 0],
    ...:          label='petal length')
    ...: plt.plot(params, weights[:, 1], linestyle='--',
    ...:          label='petal width')
    ...: plt.ylabel('weight coefficient')
    ...: plt.xlabel('C')
    ...: plt.legend(loc='upper left')
    ...: plt.xscale('log')
    ...: # plt.savefig('./figures/regression_path.png', dpi=300)
    ...: plt.show()
    ...:
```

- As we decrease C, i.e. increase regularization strength (lambda), the weight coefficients shrink – their absolute value decreases (i.e. less overfitting)

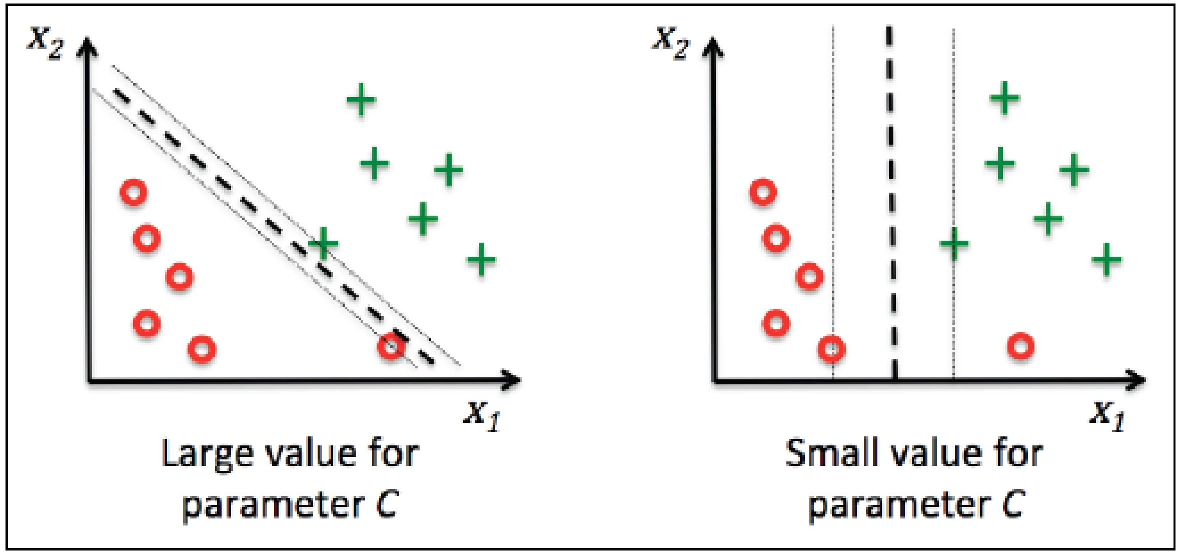


Support Vector Machines (SVM)

- Extension of perceptron
- In perceptron, minimize misclassification error
- SVM: optimization objective is to maximize the margin
  - Distance between decision boundary (“separating hyperplane”) and training samples closest to this hyperplane (the “support vectors”)

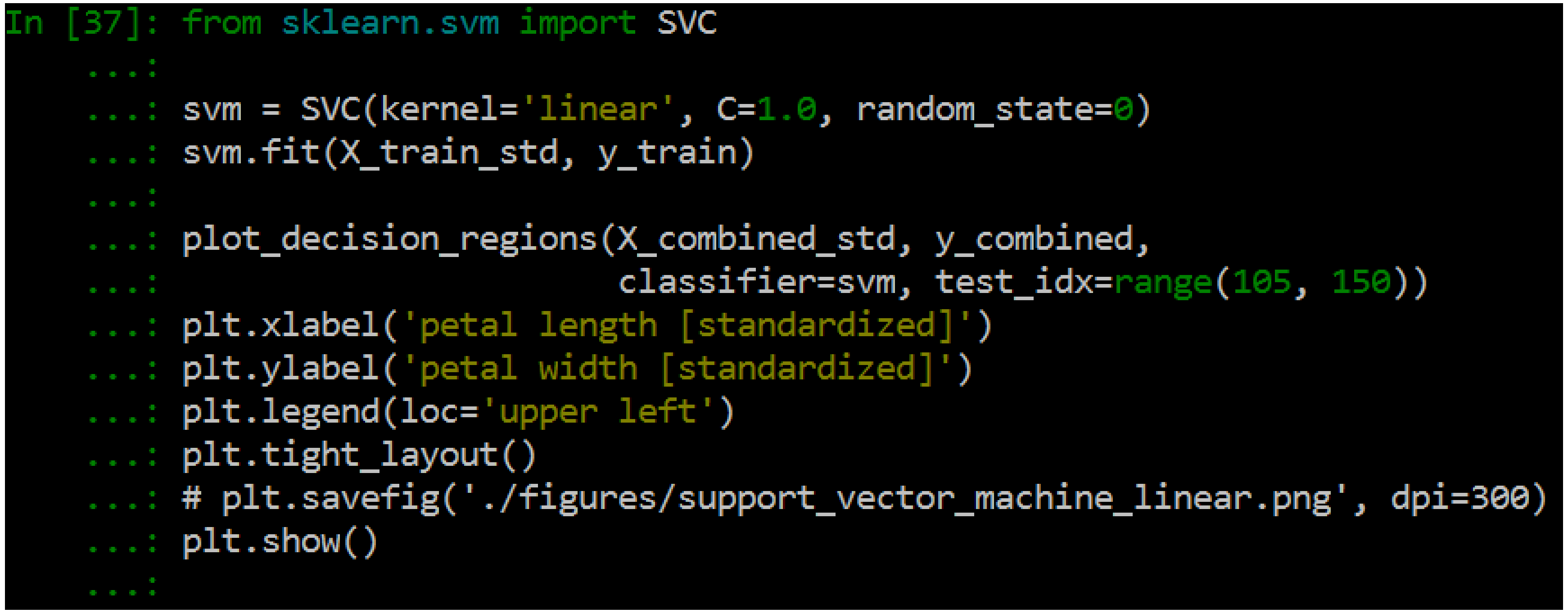


Non-linearly separable case – slack variables



- Variable "C" – control penalty for misclassification
- Large "C" – large error penalties for misclassification
- Small "C" – small error penalties for misclassification – when we are less strict about misclassification errors
- Decreasing "C" (inverse of regularization constant) increases bias and lowers variance of model

$$C = \frac{1}{\lambda}$$



```
In [37]: from sklearn.svm import SVC
    ...:
    ...: svm = SVC(kernel='linear', C=1.0, random_state=0)
    ...: svm.fit(X_train_std, y_train)
    ...:
    ...: plot_decision_regions(X_combined_std, y_combined,
    ...:                       classifier=svm, test_idx=range(105, 150))
    ...: plt.xlabel('petal length [standardized]')
    ...: plt.ylabel('petal width [standardized]')
    ...: plt.legend(loc='upper left')
    ...: plt.tight_layout()
    ...: # plt.savefig('./figures/support_vector_machine_linear.png', dpi=300)
    ...: plt.show()
    ...:
```

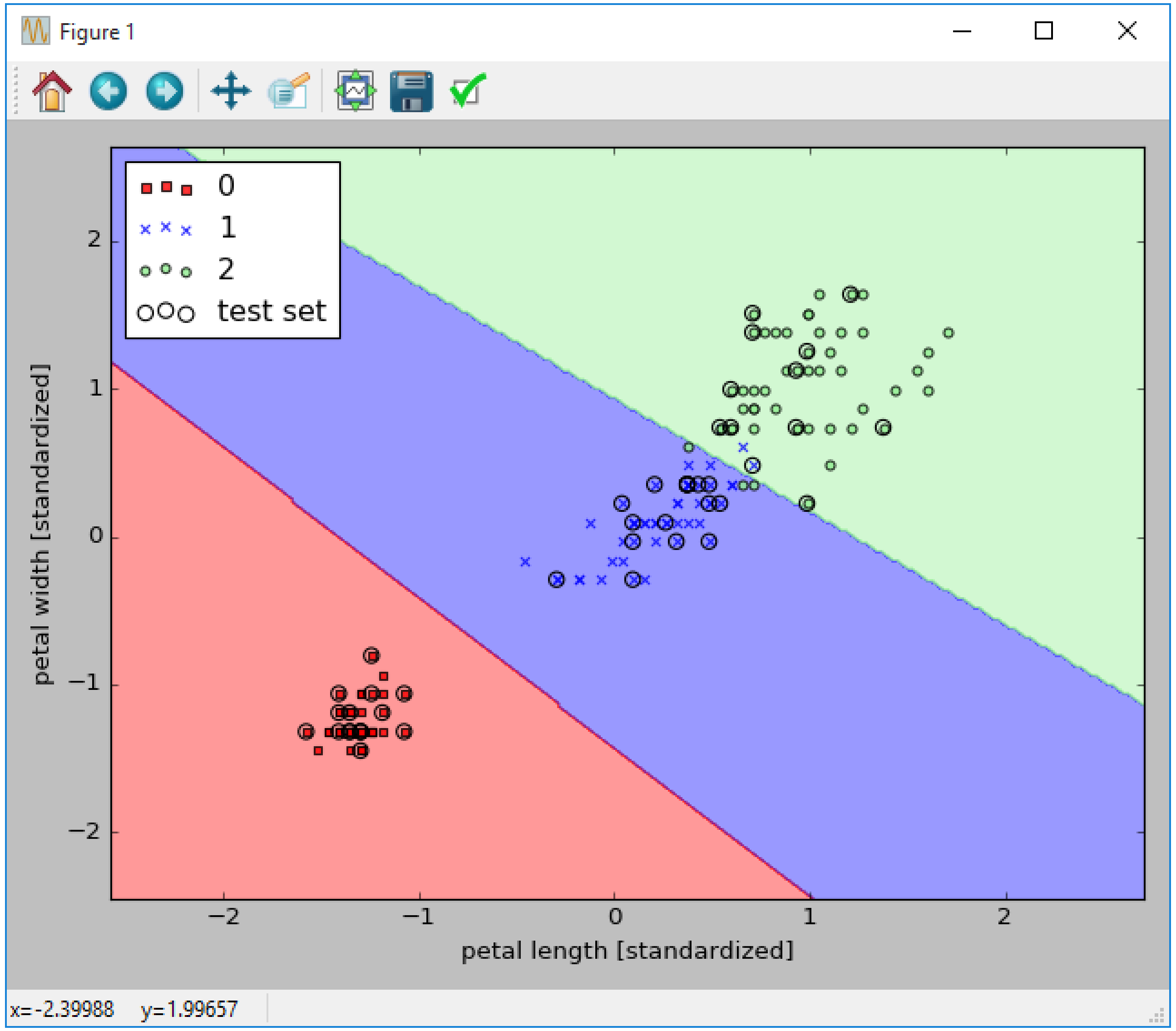
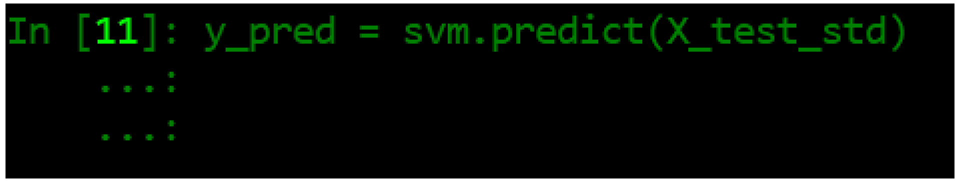
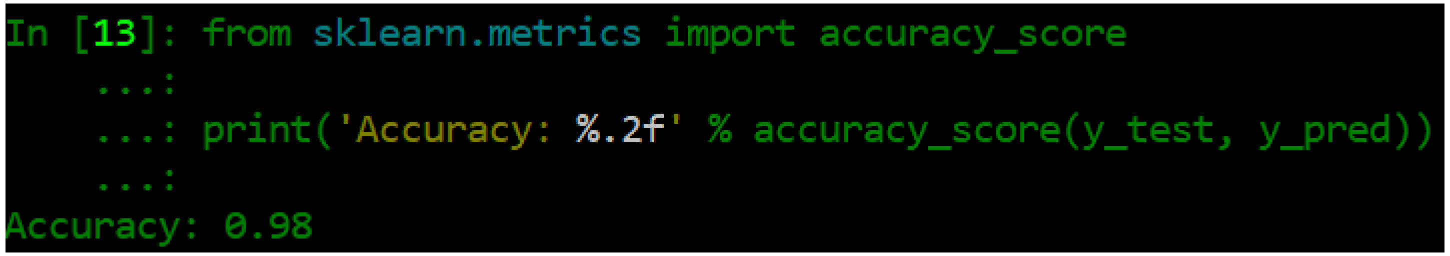


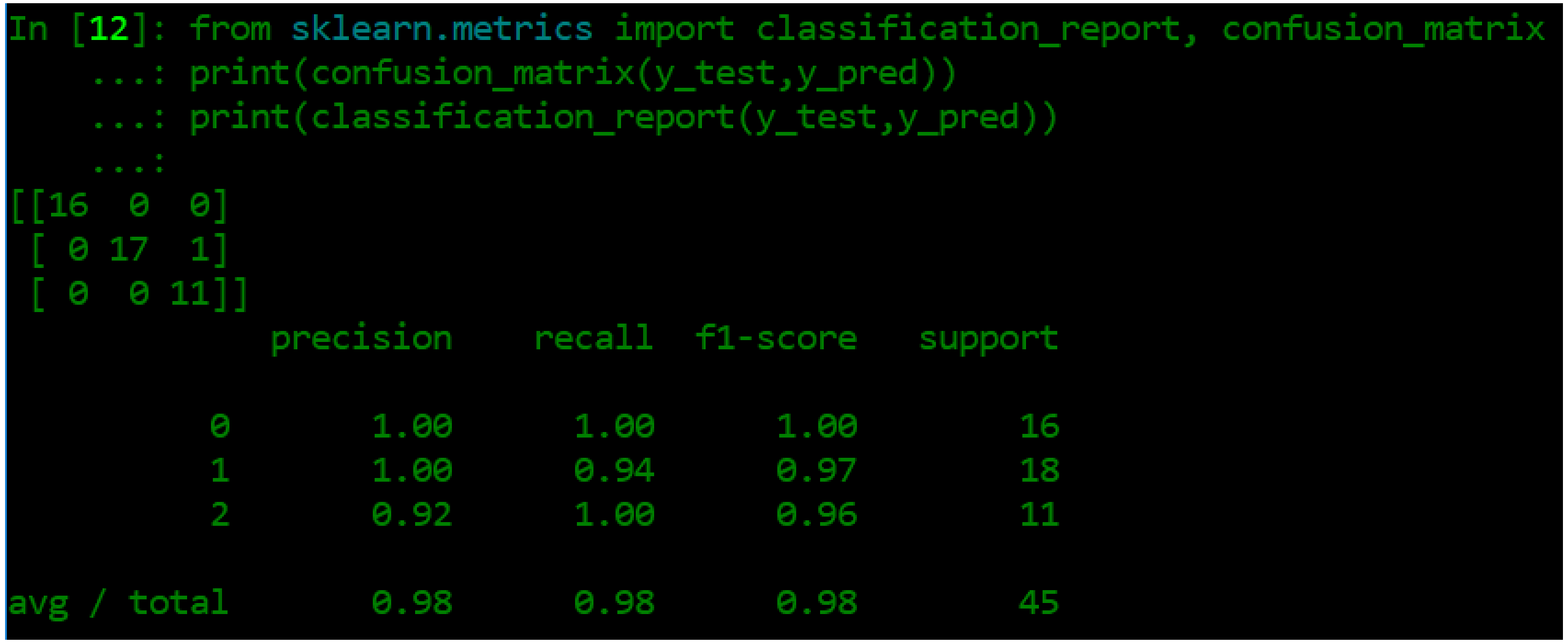
Figure 1
0
1
2
test set
petal width [standardized]
petal length [standardized]
2
1
0
−1
−2
−2
−1
0
1
2
x=-2.39988 y=1.99657



```
In [11]: y_pred = svm.predict(X_test_std)
    ...:
    ...:
```



```
In [13]: from sklearn.metrics import accuracy_score
    ...:
    ...: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
    ...:
Accuracy: 0.98
```

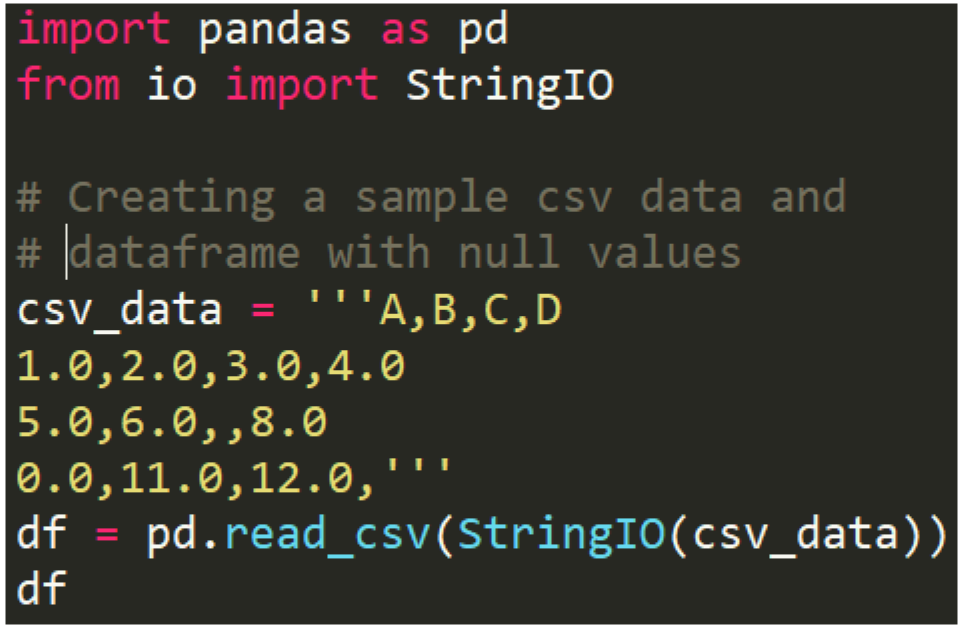


```
In [12]: from sklearn.metrics import classification_report, confusion_matrix
    ...: print(confusion_matrix(y_test,y_pred))
    ...: print(classification_report(y_test,y_pred))
    ...:
[[16  0  0]
 [ 0 17  1]
 [ 0  0 11]]
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        16
          1       1.00      0.94      0.97        18
          2       0.92      1.00      0.96        11

avg / total       0.98      0.98      0.98        45
```

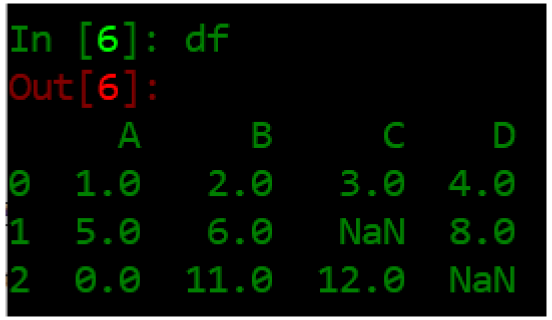
Logistic regression vs. SVM

- Linear logistic regression (LR) and linear SVM – similar results
- LR more prone to outliers than SVM
- SVM mostly cares about points closest to decision boundary (support vectors)
- LR is simpler model, implemented more easily
- LR is more easily updated – advantage with streaming data

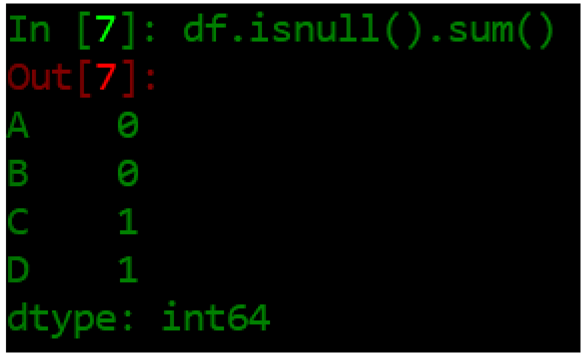
Data Preprocessing



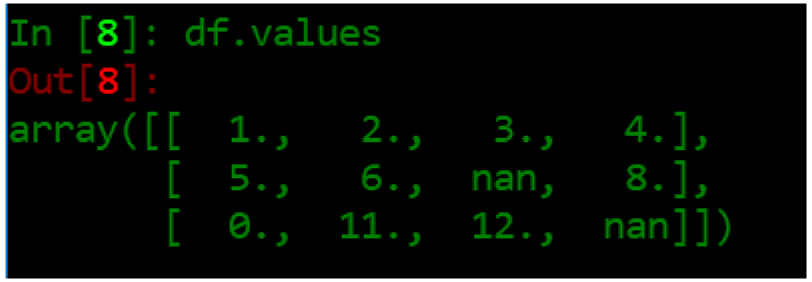
```
import pandas as pd
from io import StringIO

# Creating a sample csv data and
# dataframe with null values
csv_data = '''A,B,C,D
1.0,2.0,3.0,4.0
5.0,6.0,,8.0
0.0,11.0,12.0,'''
df = pd.read_csv(StringIO(csv_data))
df
```



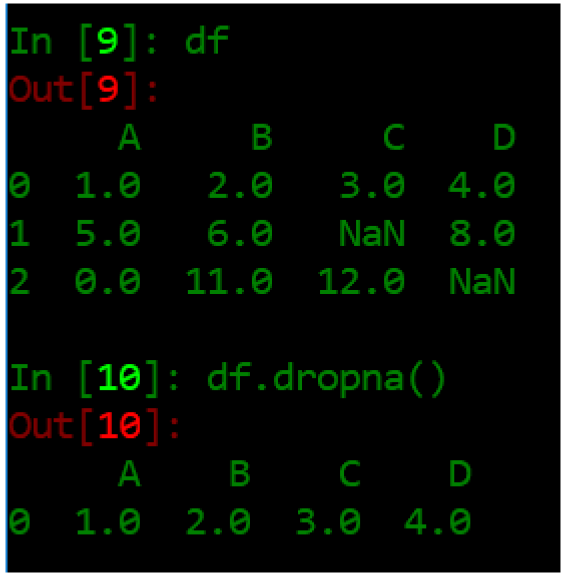
```
In [6]: df
Out[6]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN
```

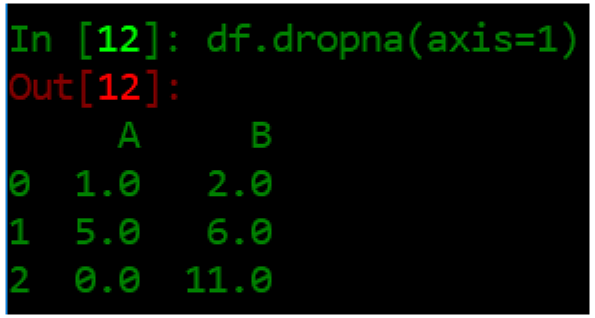


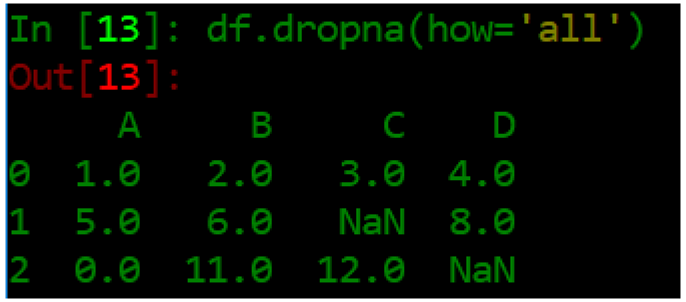
```
In [7]: df.isnull().sum()
Out[7]:
A    0
B    0
C    1
D    1
dtype: int64
```

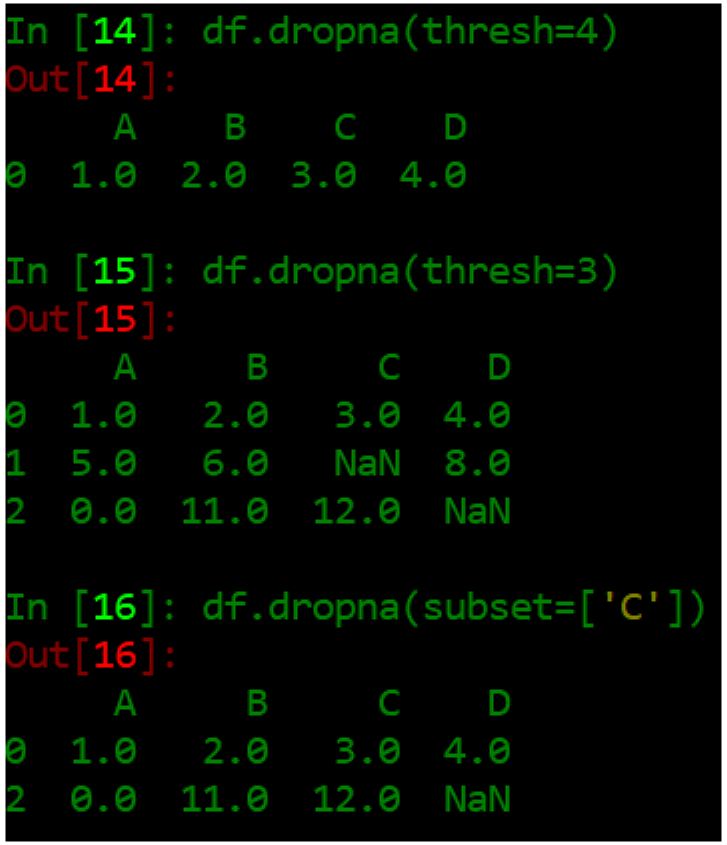


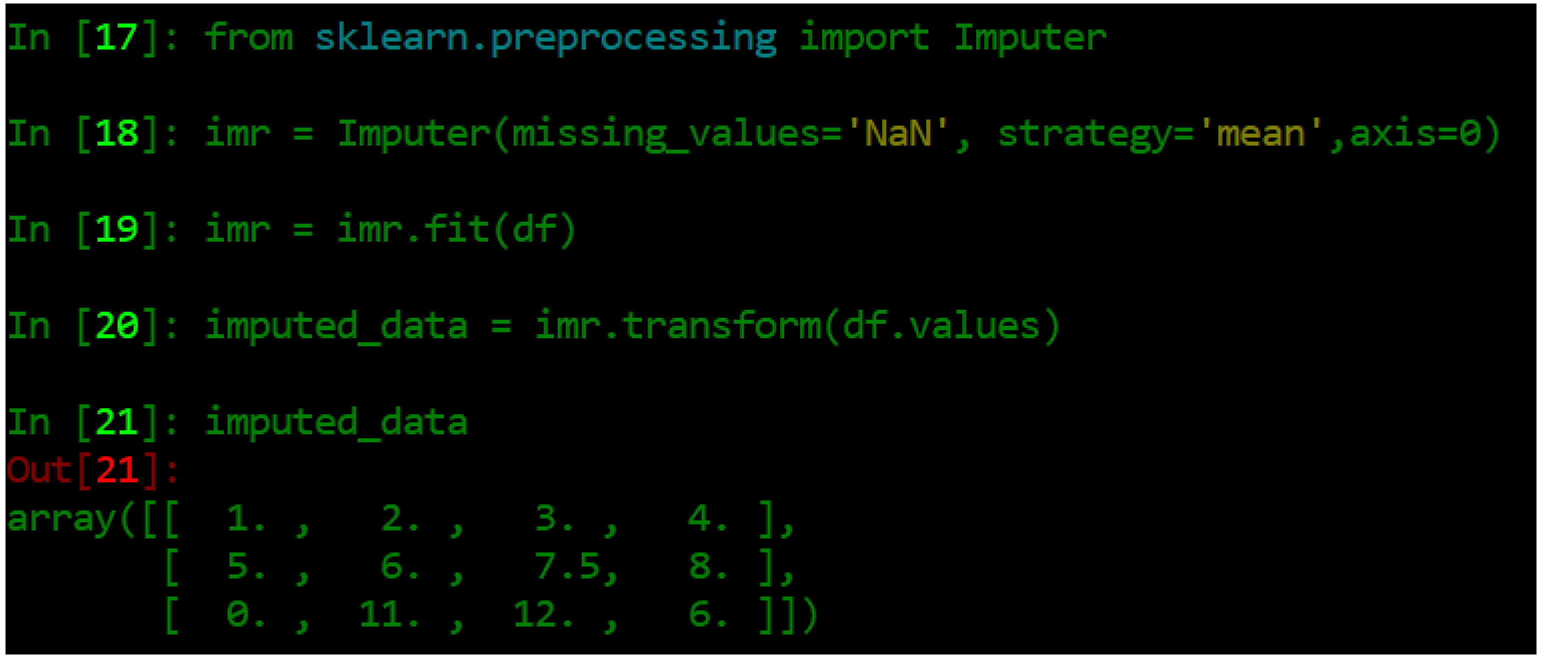
```
In [8]: df.values
Out[8]:
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [  0.,  11.,  12.,  nan]])
```


```
In [9]: df
Out[9]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN

In [10]: df.dropna()
Out[10]:
     A    B    C    D
0  1.0  2.0  3.0  4.0
```

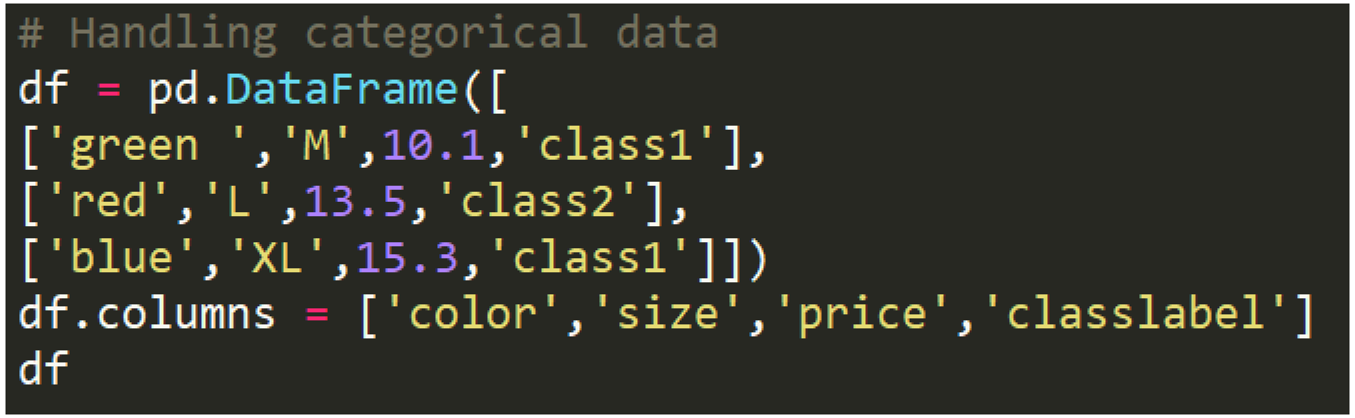

```
In [12]: df.dropna(axis=1)
Out[12]:
     A     B
0  1.0   2.0
1  5.0   6.0
2  0.0  11.0
```


```
In [13]: df.dropna(how='all')
Out[13]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN
```

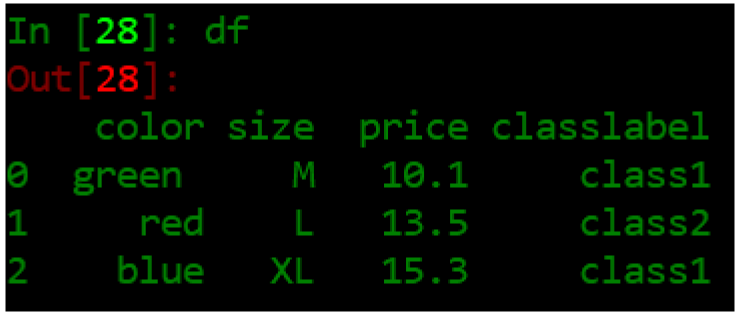

```
In [14]: df.dropna(thresh=4)
Out[14]:
     A    B    C    D
0  1.0  2.0  3.0  4.0

In [15]: df.dropna(thresh=3)
Out[15]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN

In [16]: df.dropna(subset=['C'])
Out[16]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
2  0.0  11.0  12.0  NaN
```



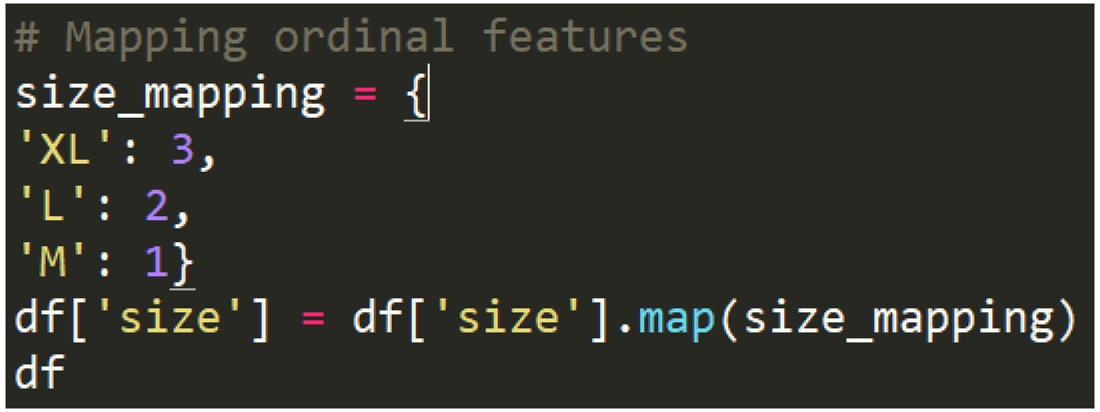
```
In [17]: from sklearn.preprocessing import Imputer

In [18]: imr = Imputer(missing_values='NaN', strategy='mean',axis=0)

In [19]: imr = imr.fit(df)

In [20]: imputed_data = imr.transform(df.values)

In [21]: imputed_data
Out[21]:
array([[  1. ,   2. ,   3. ,   4. ],
       [  5. ,   6. ,   7.5,   8. ],
       [  0. ,  11. ,  12. ,   6. ]])
```



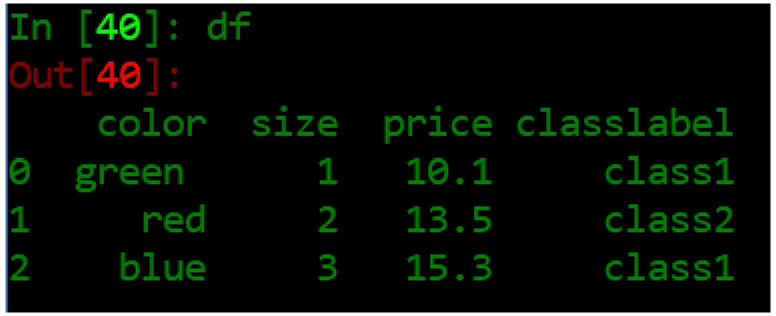
```python
# Handling categorical data
df = pd.DataFrame([
['green ','M',10.1,'class1'],
['red','L',13.5,'class2'],
['blue','XL',15.3,'class1']])
df.columns = ['color','size','price','classlabel']
df
```



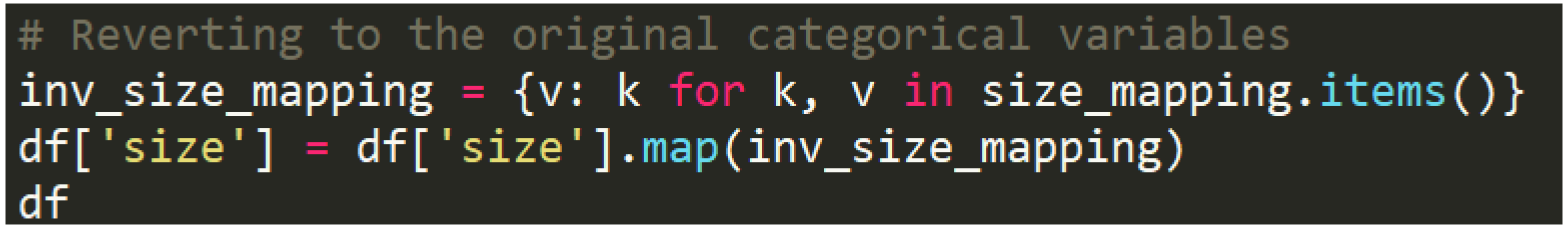
```
In [28]: df
Out[28]:
    color size  price classlabel
0  green     M   10.1     class1
1     red    L   13.5     class2
2    blue   XL   15.3     class1
```



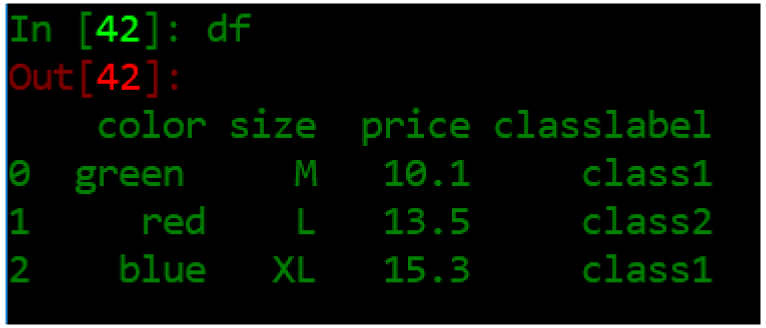
```python
# Mapping ordinal features
size_mapping = {
'XL': 3,
'L': 2,
'M': 1}
df['size'] = df['size'].map(size_mapping)
df
```



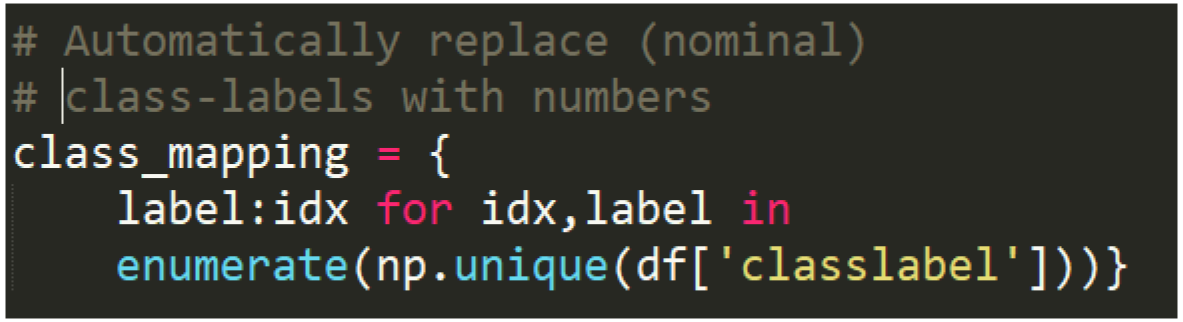
```
In [40]: df
Out[40]:
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```



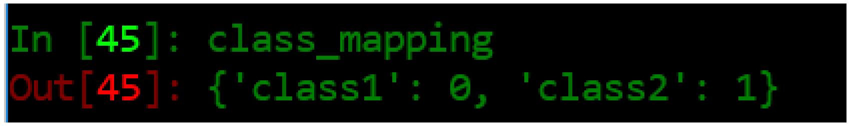
```python
# Reverting to the original categorical variables
inv_size_mapping = {v: k for k, v in size_mapping.items()}
df['size'] = df['size'].map(inv_size_mapping)
df
```



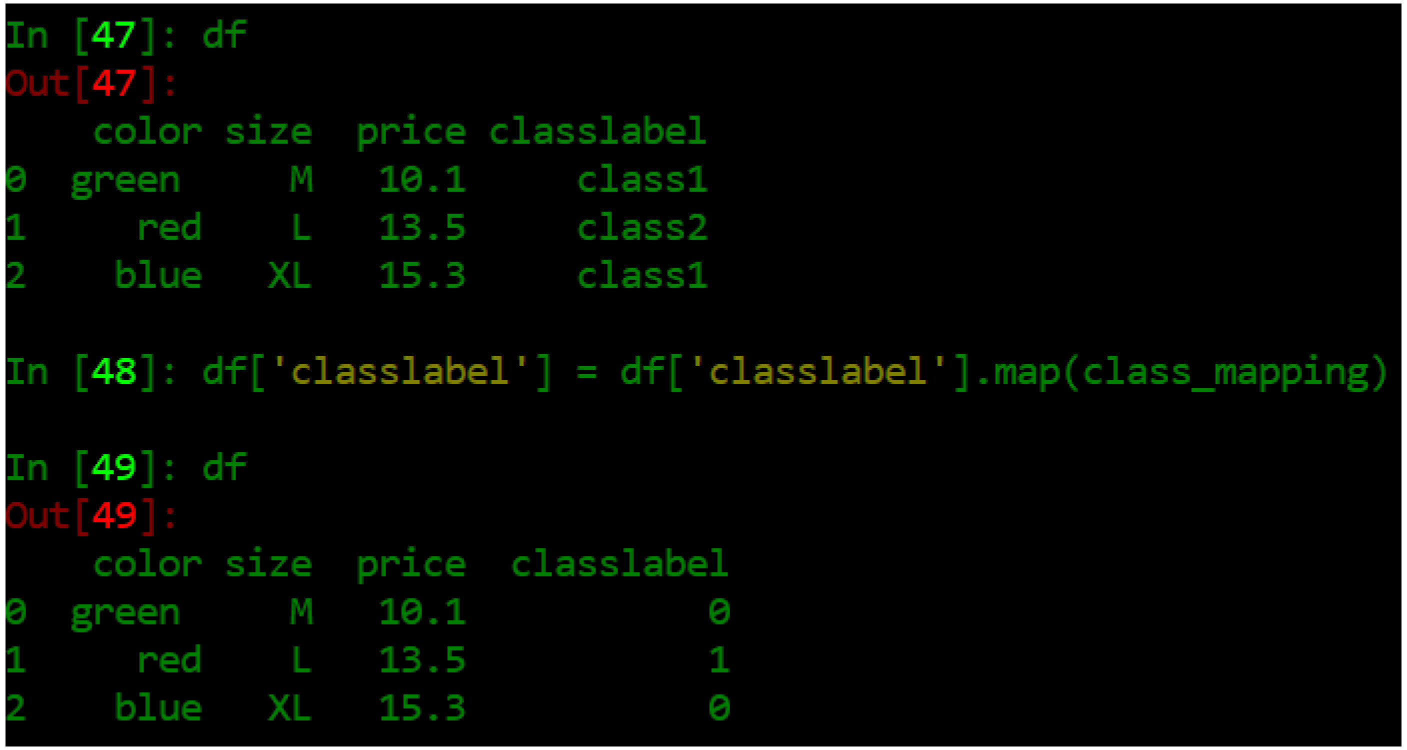
```
In [42]: df
Out[42]:
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```



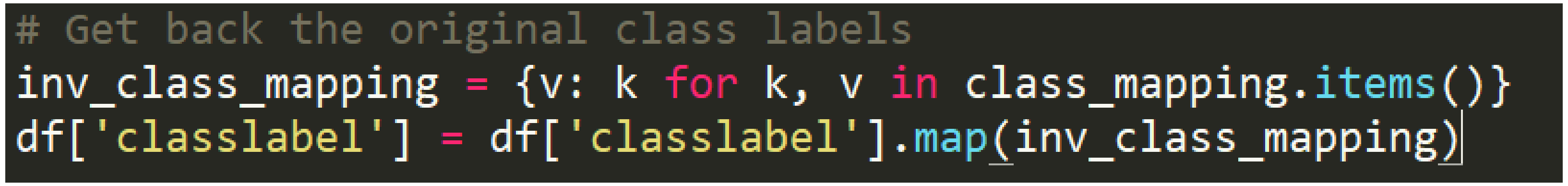
```
# Automatically replace (nominal)
# class-labels with numbers
class_mapping = {
    label:idx for idx,label in
    enumerate(np.unique(df['classlabel']))}
```



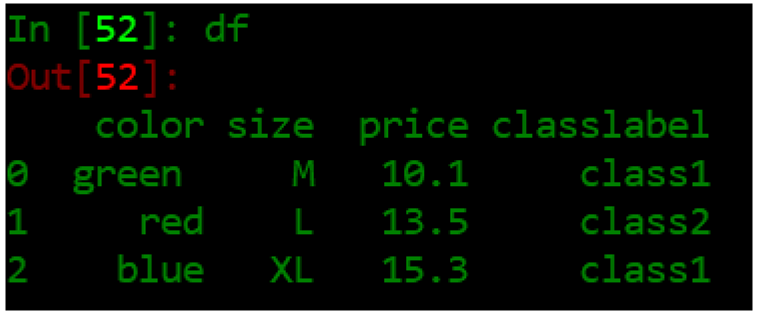
```
In [45]: class_mapping
Out[45]: {'class1': 0, 'class2': 1}
```



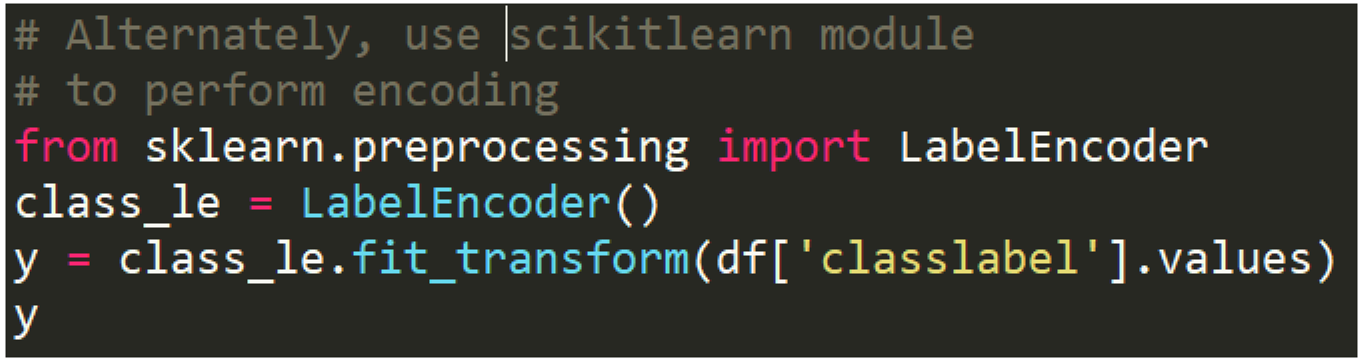
```
In [47]: df
Out[47]:
    color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1

In [48]: df['classlabel'] = df['classlabel'].map(class_mapping)

In [49]: df
Out[49]:
    color size  price  classlabel
0  green    M   10.1           0
1    red    L   13.5           1
2   blue   XL   15.3           0
```



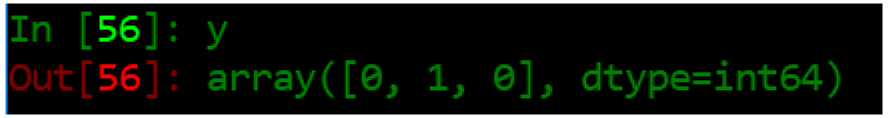
```
# Get back the original class labels
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
```



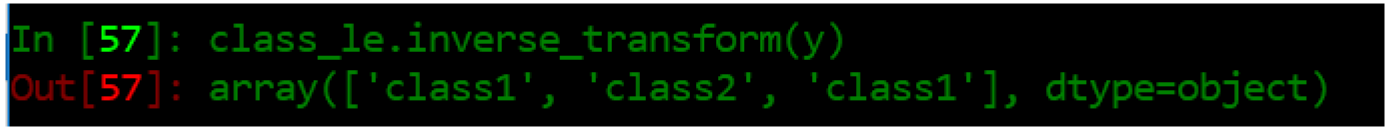
```
In [52]: df
Out[52]:
    color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```



```
# Alternately, use scikitlearn module
# to perform encoding
from sklearn.preprocessing import LabelEncoder
class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
y
```



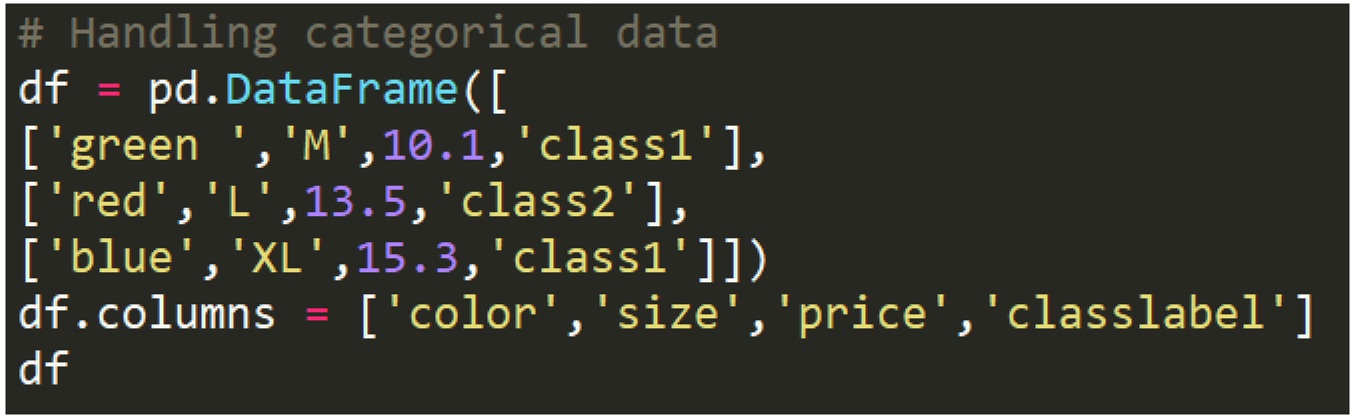
```
In [56]: y
Out[56]: array([0, 1, 0], dtype=int64)
```



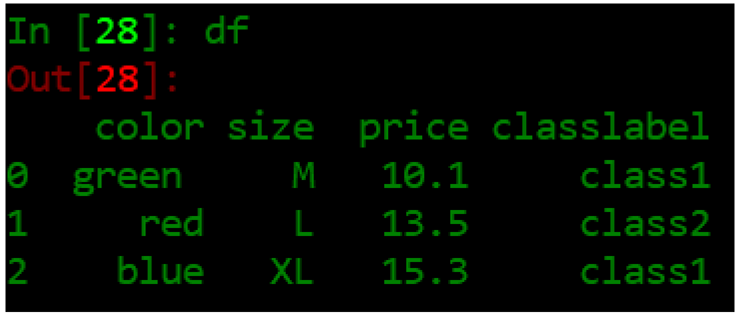
```
In [57]: class_le.inverse_transform(y)
Out[57]: array(['class1', 'class2', 'class1'], dtype=object)
```

One-hot encoding

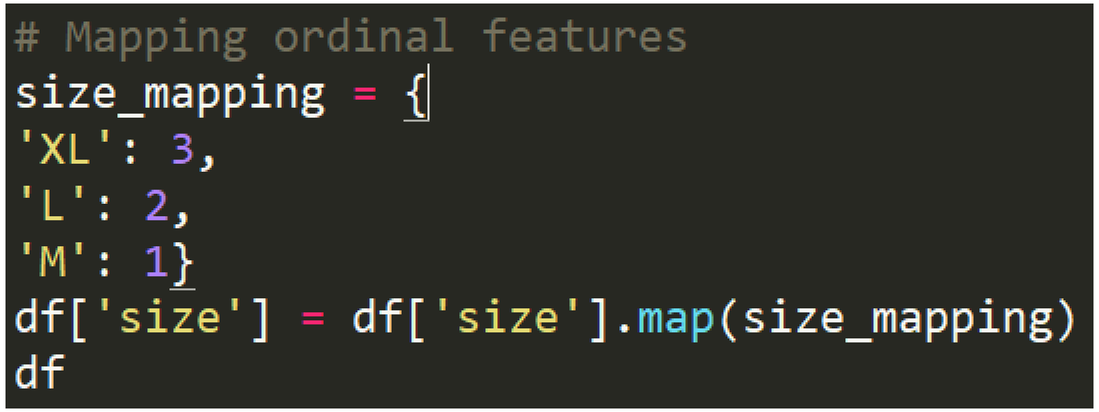
- Problem with the following approach: order implied by number



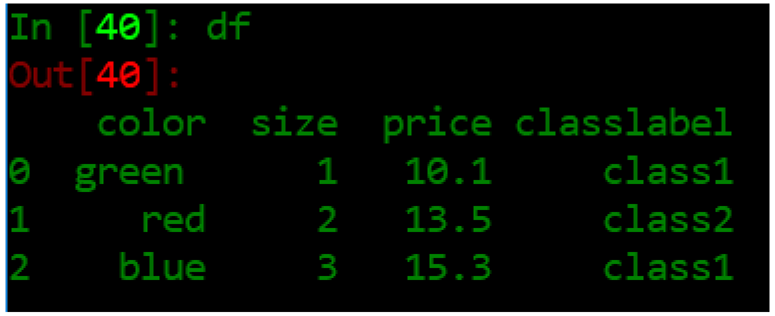
```
# Handling categorical data
df = pd.DataFrame([
['green ','M',10.1,'class1'],
['red','L',13.5,'class2'],
['blue','XL',15.3,'class1']])
df.columns = ['color','size','price','classlabel']
df
```



```
In [28]: df
Out[28]:
    color size  price classlabel
0  green     M   10.1     class1
1     red    L   13.5     class2
2    blue   XL   15.3     class1
```

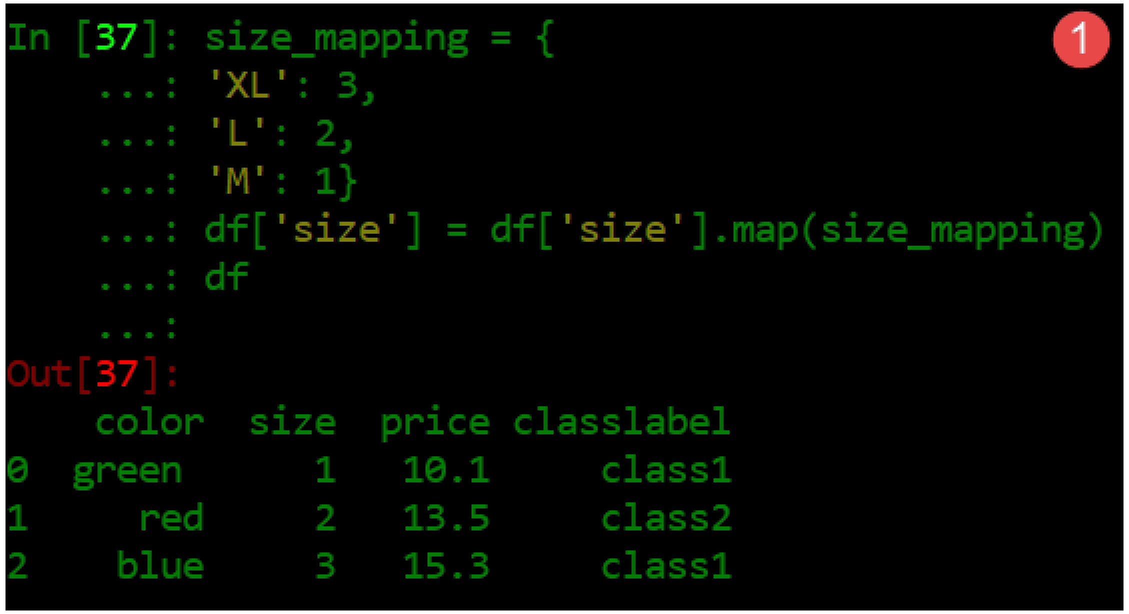


```
# Mapping ordinal features
size_mapping = {
'XL': 3,
'L': 2,
'M': 1}
df['size'] = df['size'].map(size_mapping)
df
```



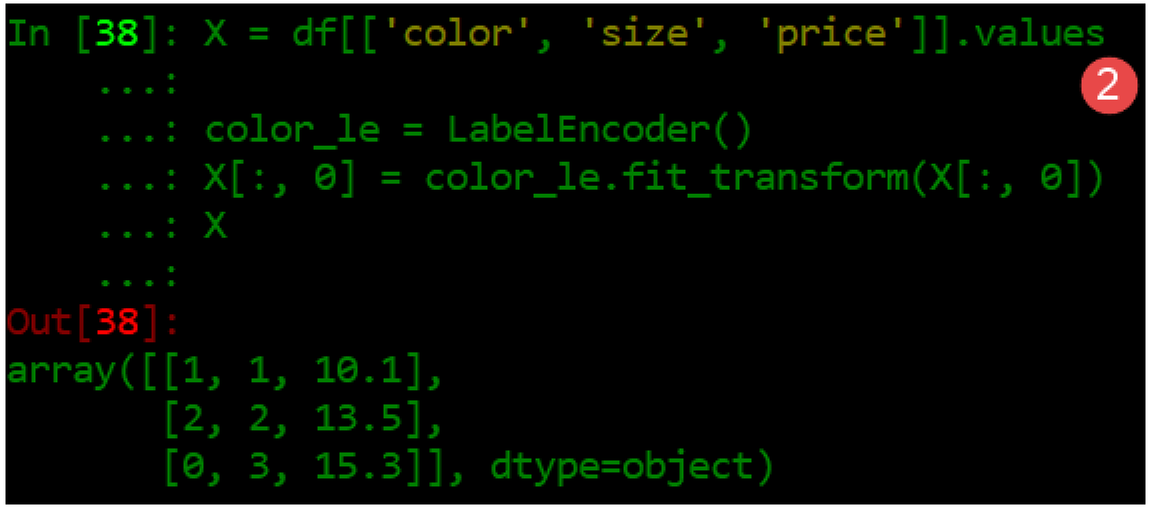
```
In [40]: df
Out[40]:
    color  size  price classlabel
0  green      1   10.1     class1
1     red     2   13.5     class2
2    blue     3   15.3     class1
```

One-hot encoding



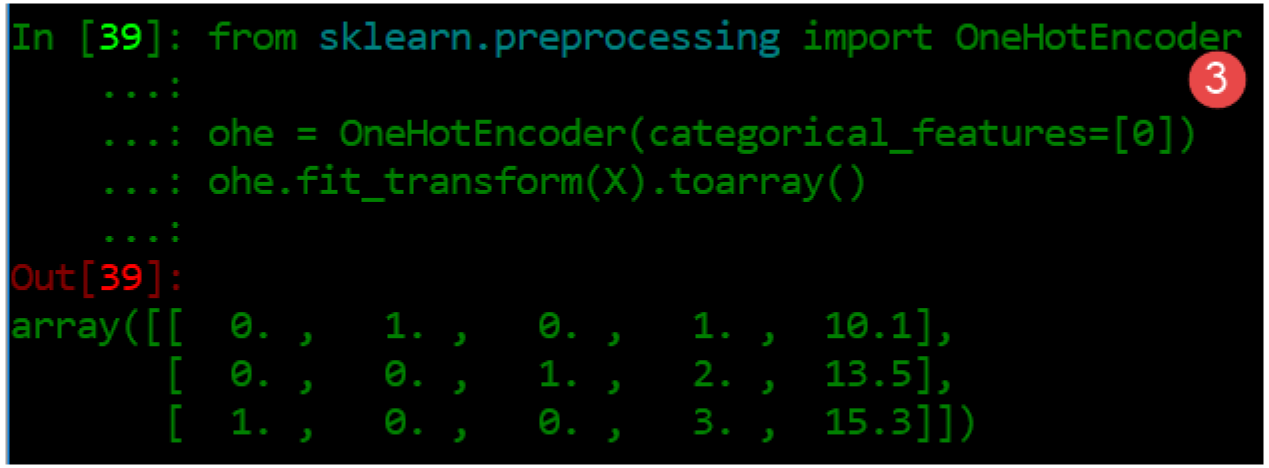
1

```
In [37]: size_mapping = {
    ...: 'XL': 3,
    ...: 'L': 2,
    ...: 'M': 1}
    ...: df['size'] = df['size'].map(size_mapping)
    ...: df
    ...:
Out[37]:
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```



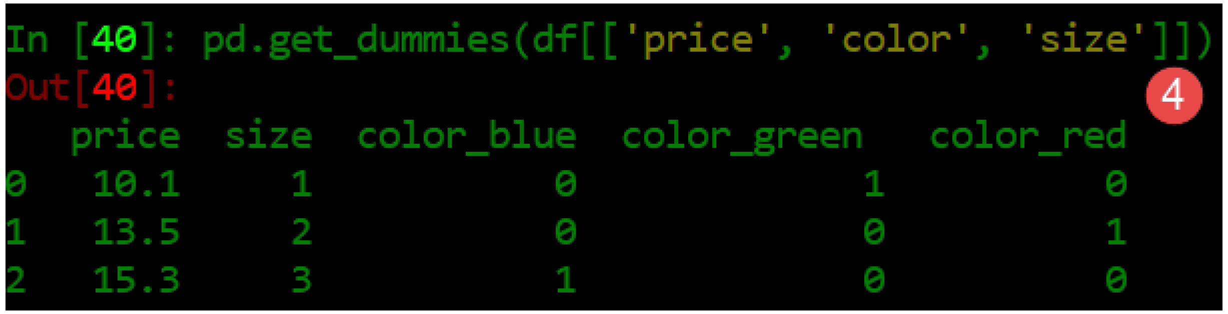
2

```
In [38]: X = df[['color', 'size', 'price']].values
    ...:
    ...: color_le = LabelEncoder()
    ...: X[:, 0] = color_le.fit_transform(X[:, 0])
    ...: X
    ...:
Out[38]:
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```



3

```
In [39]: from sklearn.preprocessing import OneHotEncoder
    ...:
    ...: ohe = OneHotEncoder(categorical_features=[0])
    ...: ohe.fit_transform(X).toarray()
    ...:
Out[39]:
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

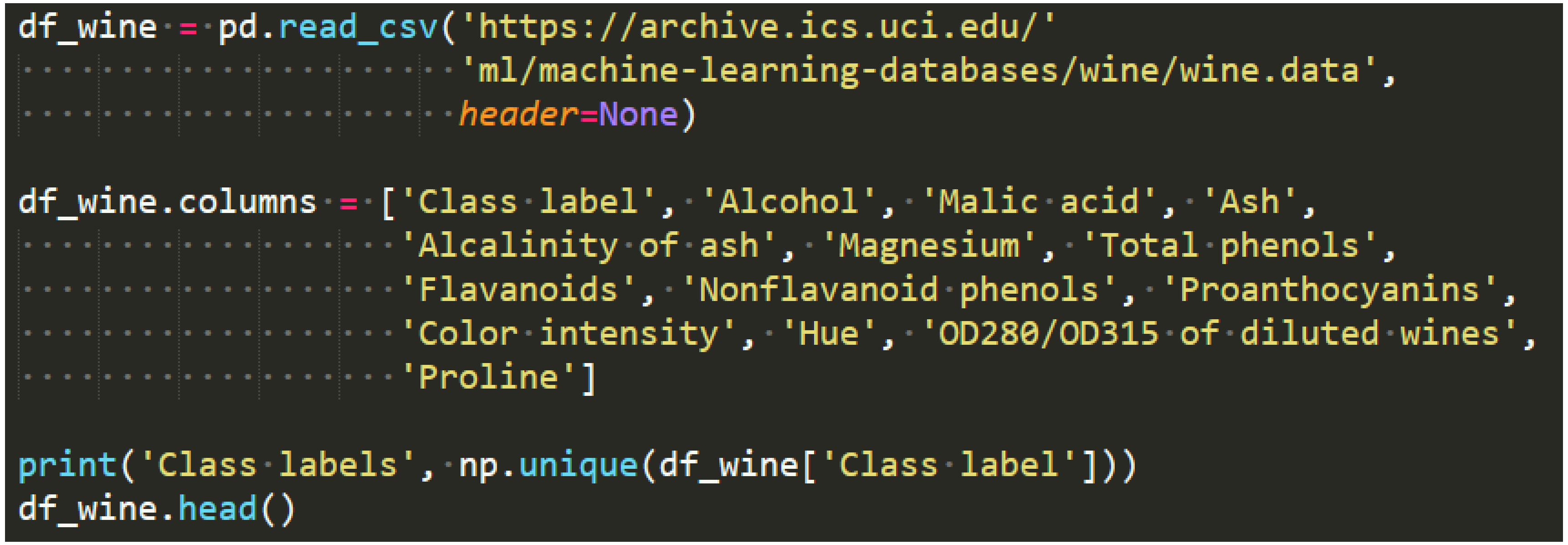


4

```
In [40]: pd.get_dummies(df[['price', 'color', 'size']])
Out[40]:
   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
```

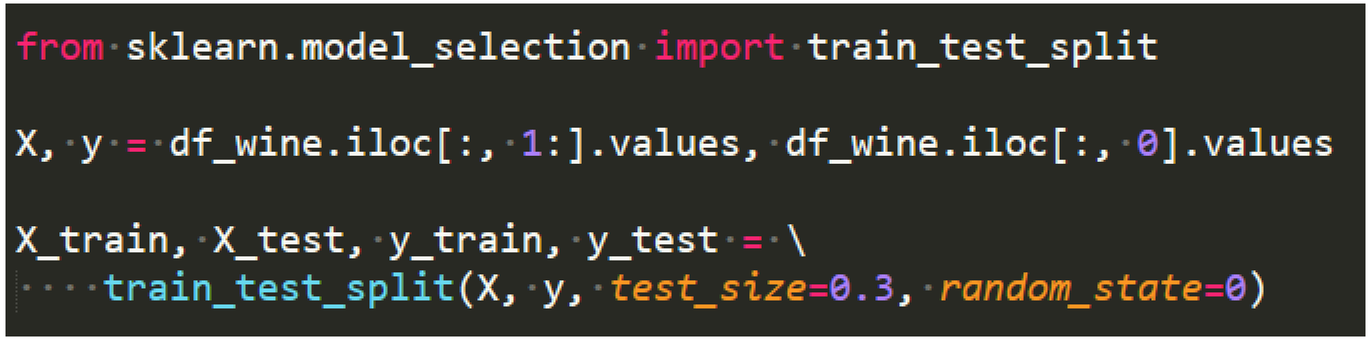
Partitioning a dataset into training and test sets

- Test set: the *ultimate* test of our model before we use it in the real world
- Import data



```python
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()
```

Train-test-split



```python
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)
```

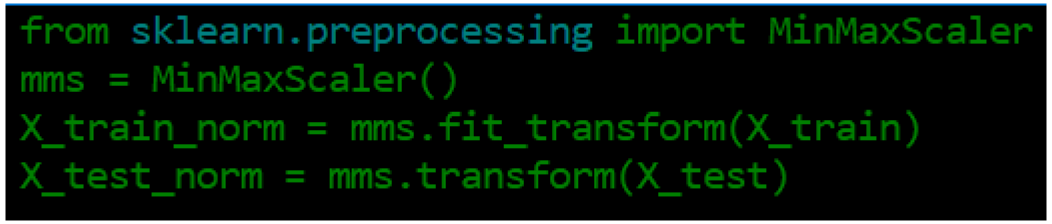
Feature scaling

- When to consider feature scaling
  - Logistic regression, SVM, perceptrons, neural networks.
  - K-nearest neighbors with a Euclidean distance measure with features contributing equally
  - K-means, Linear discriminant analysis, principal component analysis, kernel principal component analysis.
  - Decision trees, random forests – no need to worry about feature scaling
- Two types of feature scaling
  - Normalization
  - Standardization

Normalization

- Normalization: bring features to [0,1] range:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$



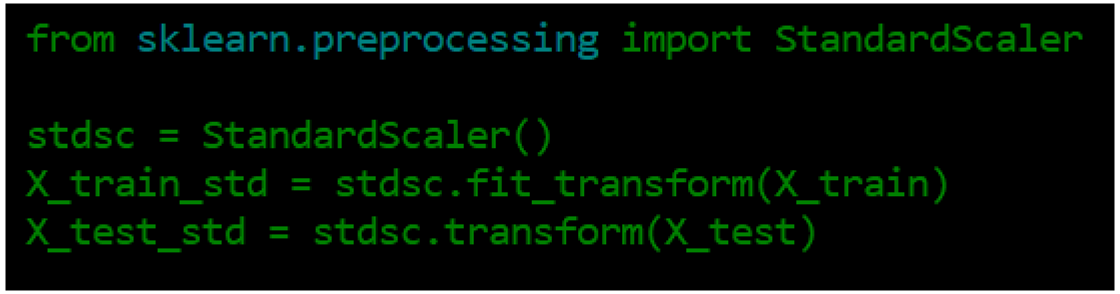
```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

- Sensitive to outliers

Standardization

- Many linear models (e.g. Logistic Regression, SVM) initialize weights to 0 or small random values close to 0
- Sometimes standardization (mean 0, std dev 1) helps learn weights better:

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



```
from sklearn.preprocessing import StandardScaler

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

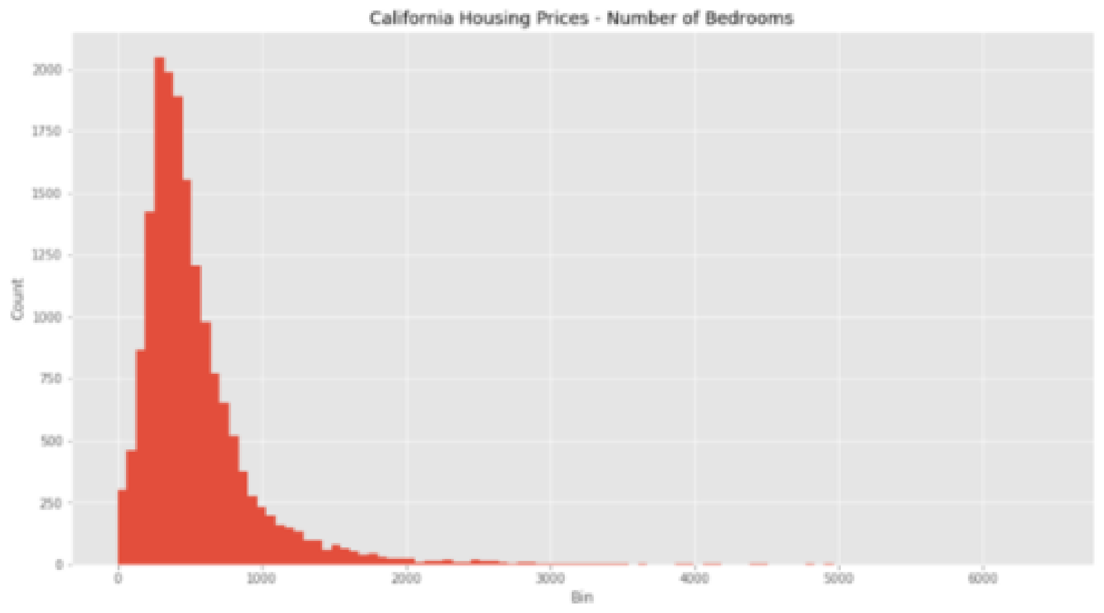
- Standardization preserves useful information about outliers
- Algorithm less sensitive to outliers than in min-max scaling

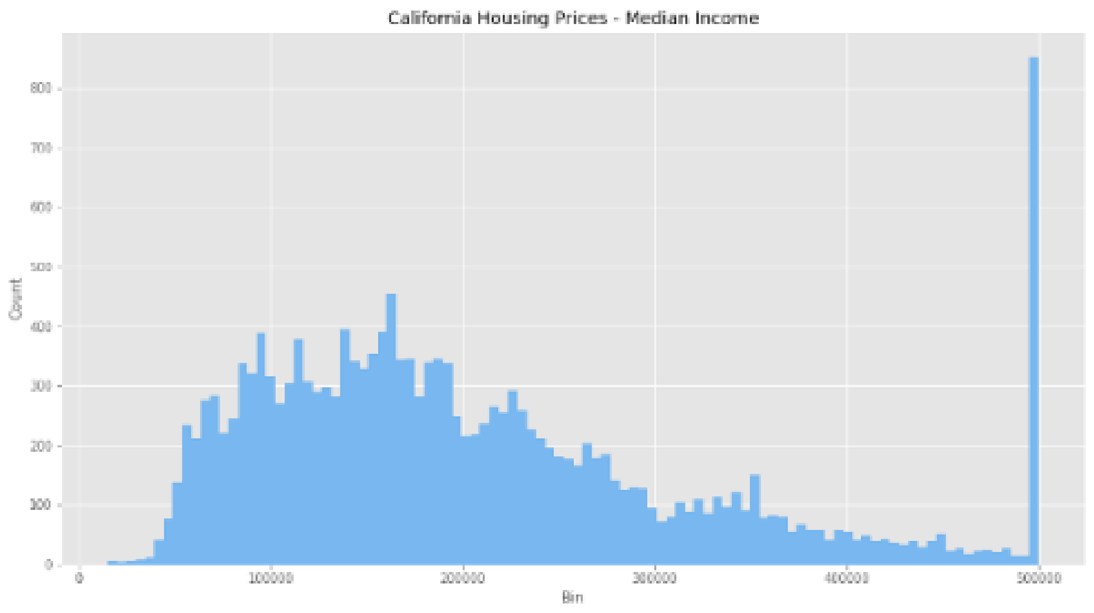
When to use what method

- Normalize when
  - distribution of data is unknown
  - distribution is known to be not Gaussian (i.e. bell curve)
  - outliers are not a concern
  - standard deviation very small
  - algorithm does not make assumptions about the distribution of data (e.g. k-nearest neighbors and artificial neural networks)
- Standardize otherwise

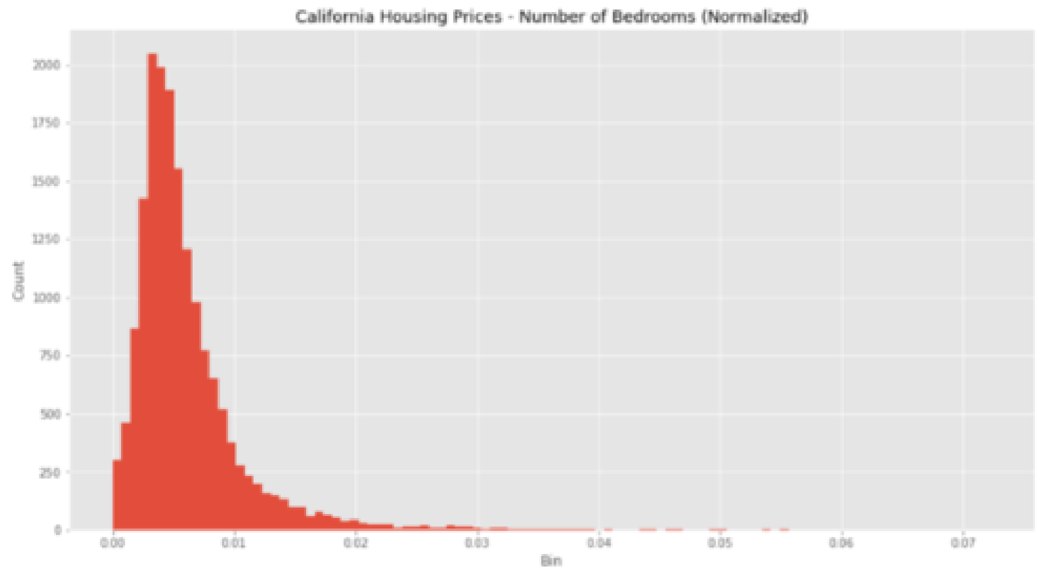
Normalization vs Standardization

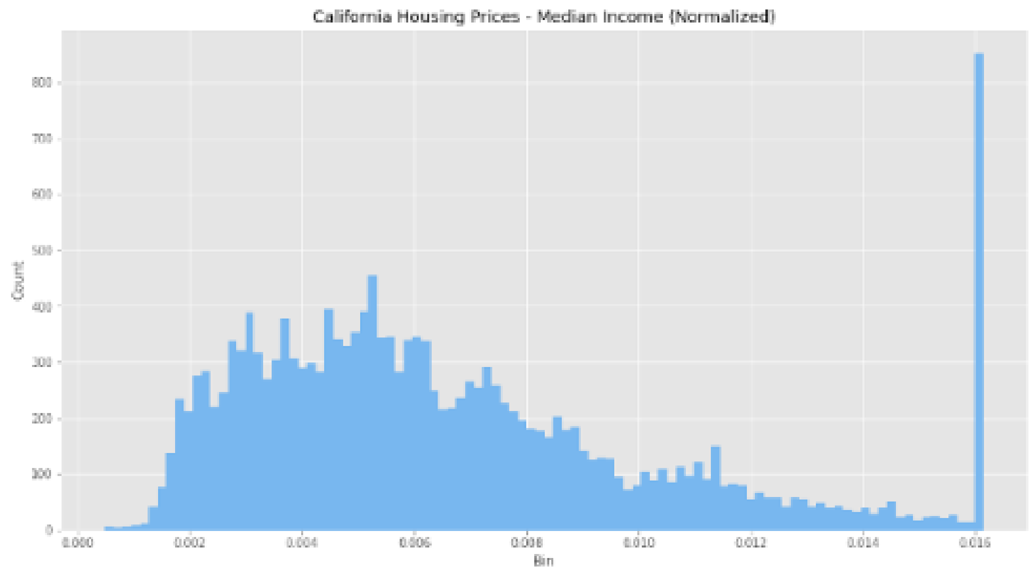
- Normalize
  - useful for optimization algorithms – e.g. gradient descent – that weight inputs (e.g. artificial neural networks).
  - also for algorithms that use distance measurements – e.g. K-Nearest-Neighbors (KNN).
- Standardize (subtract from mean, divide by std dev)
  - linear regression, logistic regression, SVM and linear discriminant analysis
- When in doubt, do both and see which works better



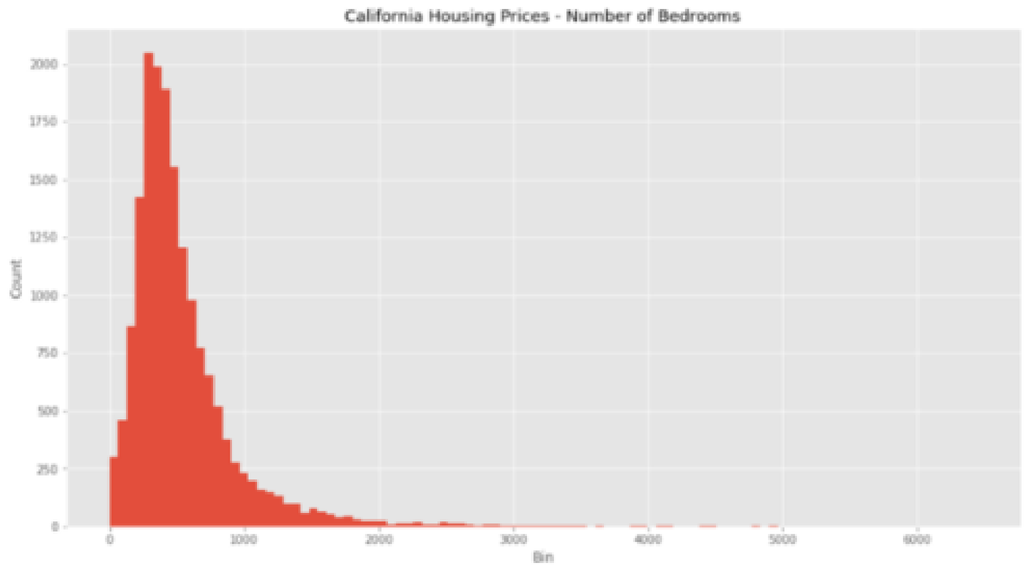


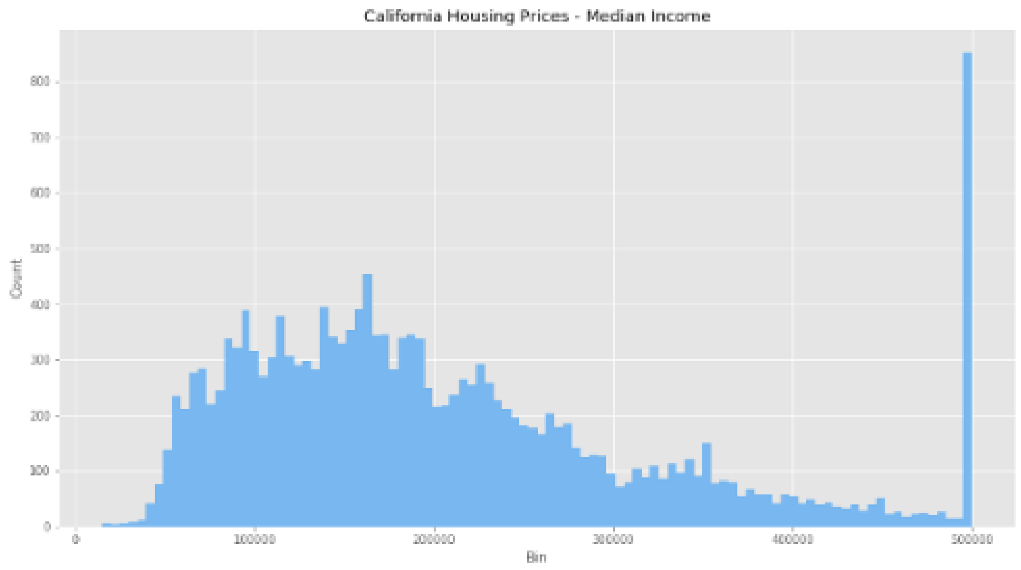
Raw Values



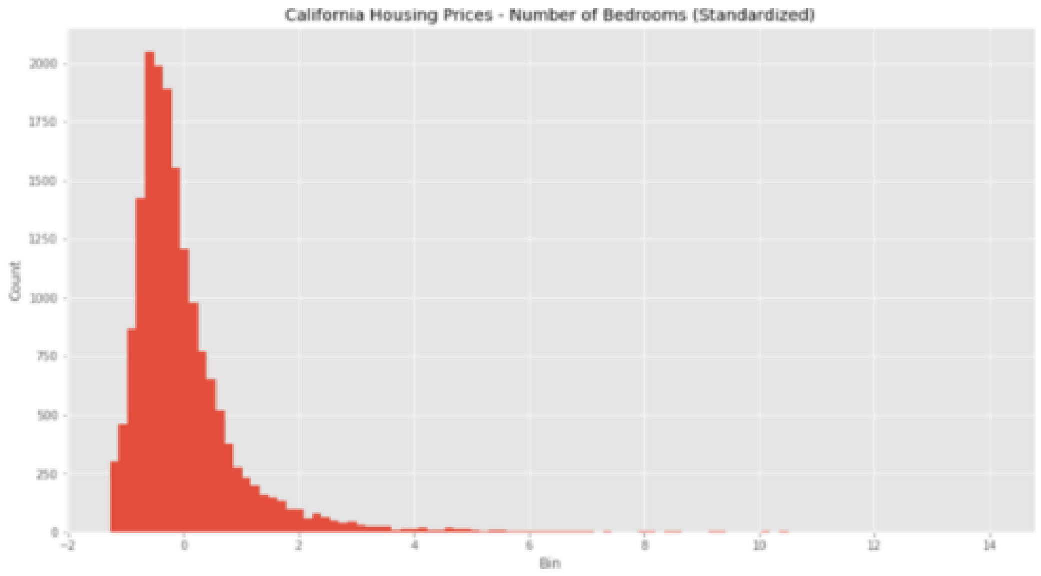


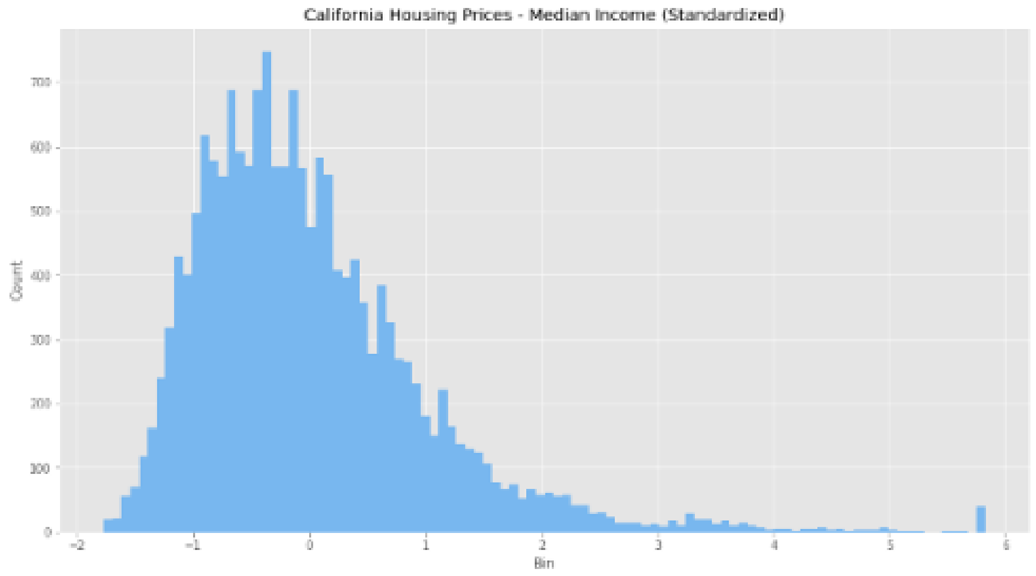
Normalized Values





Raw Values

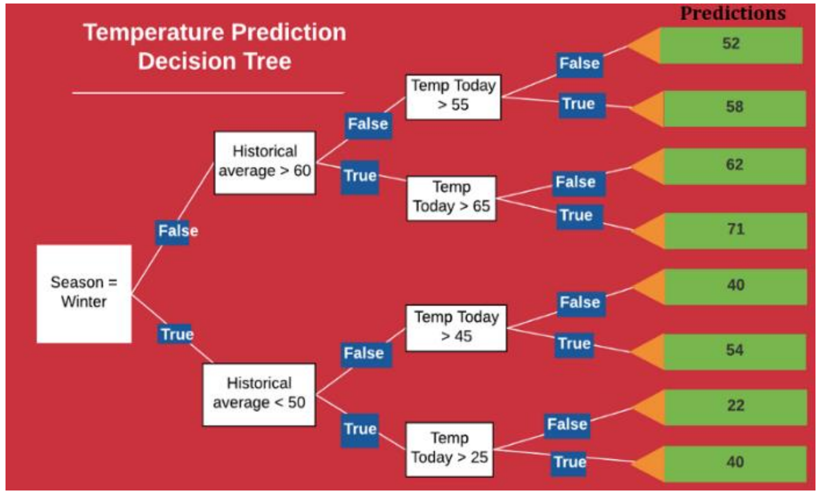




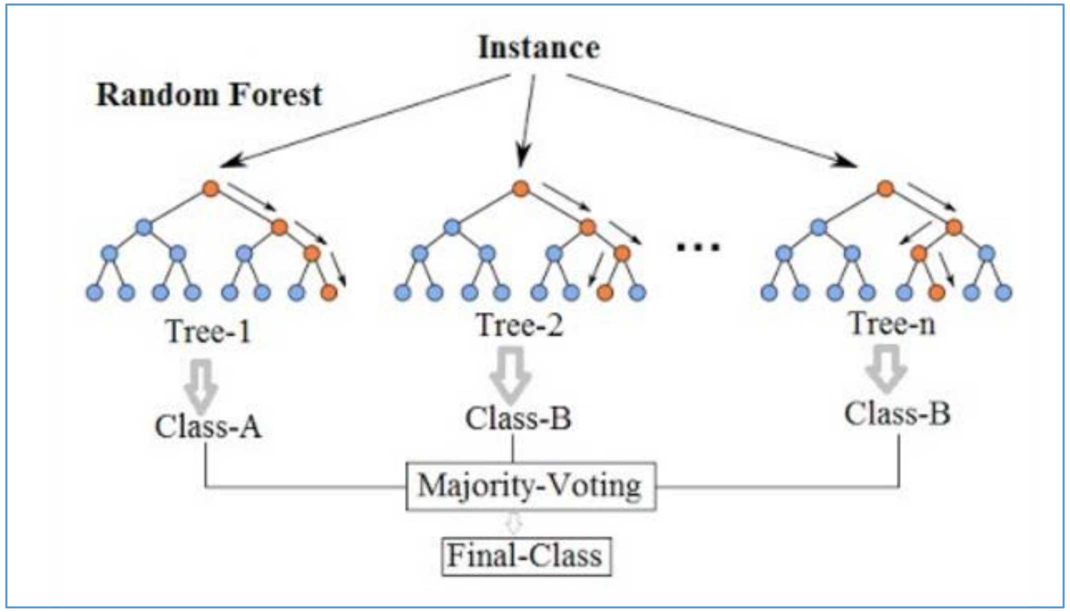
Standardized Values

Random forest – brief overview

- One decision tree:



- Random forest: many decision trees – on random subsets of training data – on random features – averaged

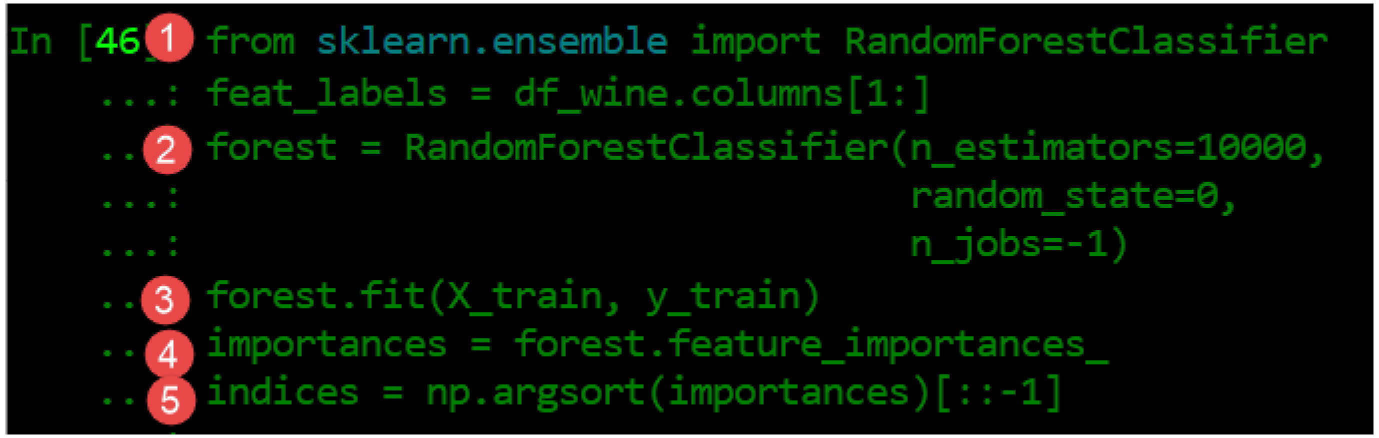


Source: https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

Assessing feature importance with Random Forests

1. Import the random forest classifier
2. Create the model, with hyper-parameters

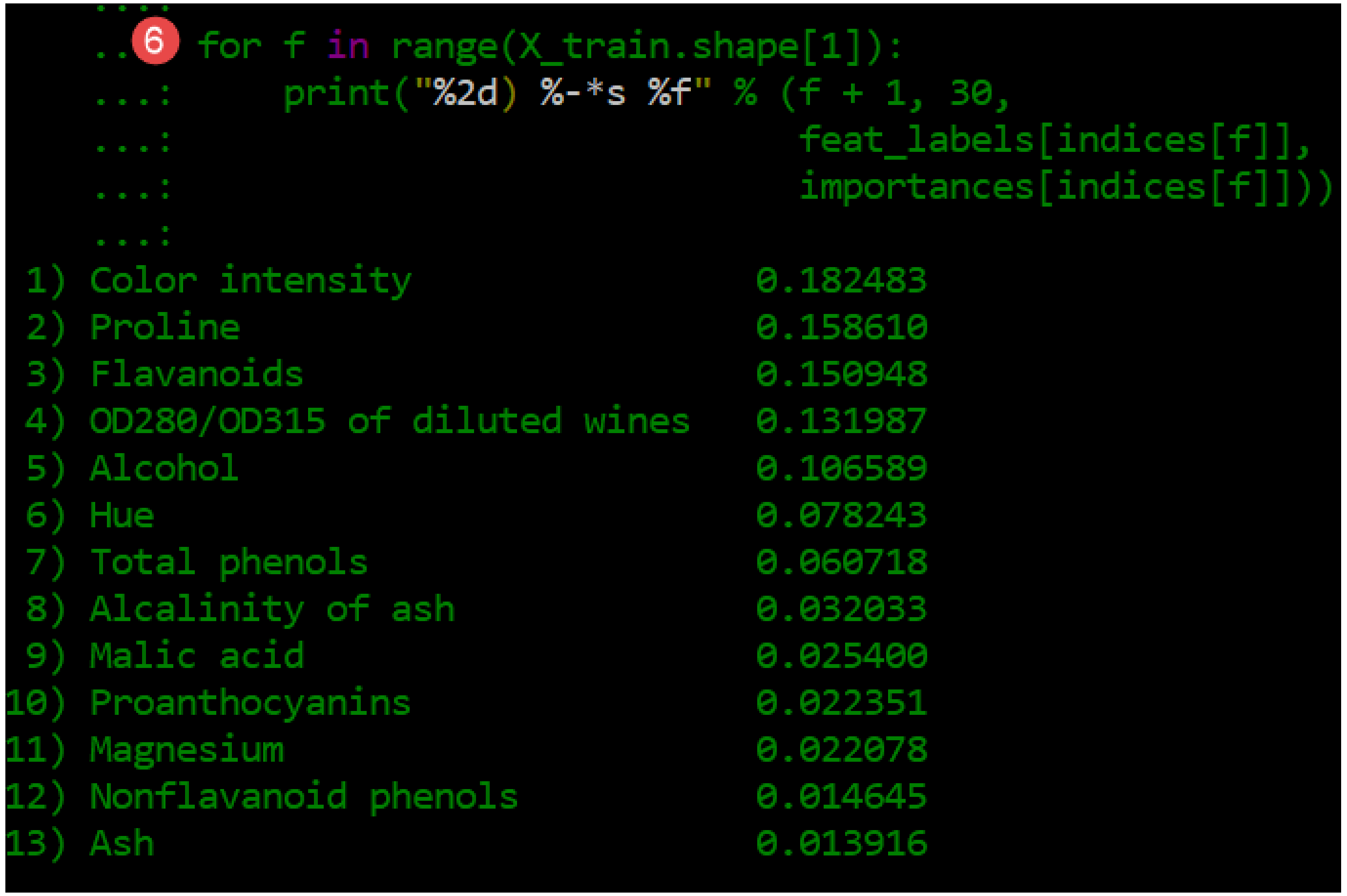
   ”n_estimators”: number of trees
3. Fit the model on your data
4. Extract feature-importances
5. Sort feature-importances



```
In [46]: from sklearn.ensemble import RandomForestClassifier
    ...: feat_labels = df_wine.columns[1:]
    ...: forest = RandomForestClassifier(n_estimators=10000,
    ...:                                 random_state=0,
    ...:                                 n_jobs=-1)
    ...: forest.fit(X_train, y_train)
    ...: importances = forest.feature_importances_
    ...: indices = np.argsort(importances)[::-1]
```

Assessing feature importance with Random Forests

6. Print all features in descending order of importance



```
    ...: for f in range(X_train.shape[1]):
    ...:     print("%2d) %-*s %f" % (f + 1, 30,
    ...:                             feat_labels[indices[f]],
    ...:                             importances[indices[f]]))
    ...:
 1) Color intensity                0.182483
 2) Proline                        0.158610
 3) Flavanoids                     0.150948
 4) OD280/OD315 of diluted wines   0.131987
 5) Alcohol                        0.106589
 6) Hue                            0.078243
 7) Total phenols                  0.060718
 8) Alcalinity of ash              0.032033
 9) Malic acid                     0.025400
10) Proanthocyanins                0.022351
11) Magnesium                      0.022078
12) Nonflavanoid phenols           0.014645
13) Ash                            0.013916
```