经济大数据与 Python 应用

Machine Learning

陈洲
湘潭大学商学院
2025

Press Space for next page →

What Is Machine Learning?

Machine learning as a means of *building models of data*

- "learning": *tunable parameters* that can be adapted to observed data

Categories of Machine Learning

Supervised learning involves somehow modeling the relationship between measured features of data and some labels associated with the data

- in *classification*, the labels are discrete categories
- in *regression*, the labels are continuous quantities.

Unsupervised learning involves modeling the features of a dataset without reference to any label.

- such as *clustering* and *dimensionality reduction.*

Introducing Scikit-Learn

An overview of the Scikit-Learn API.

```
conda install scikit-learn -c conda-forge
```

Data Representation in Scikit-Learn

Table is a two-dimensional grid of data

- rows: individual elements of the dataset,
- columns: quantities related to each of these elements

For example:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

or `pd.read_csv` iris.csv

The Features Matrix

this matrix is often stored in a variable named `X`

The Target Array

by convention we will usually call `y`.

For use in Scikit-Learn,

- we will extract the features matrix and target array from the `DataFrame`

```
X_iris = iris.drop('species', axis=1)
X_iris.shape
```

```
y_iris = iris['species']
y_iris.shape
```

Supervised Learning Example: Simple Linear Regression

Fitting a line to (x, y) data.

Generate data

```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y);
```

1. Choose a class of model

In Scikit-Learn,

- to compute a simple `LinearRegression` model
- import the linear regression class:

```
from sklearn.linear_model import LinearRegression
```

2. Choose model hyperparameters

a class of model is not the same as an instance of a model

Some options open to us.

- Would we like to fit for the offset (i.e., *y*-intercept)?
- Would we like to preprocess our features to add model flexibility?
- How many model components would we like to use?

These choices are often represented as *hyperparameters* or parameters that must be set before the model is fit to data.

Fit the intercept using the `fit_intercept` hyperparameter:

```
model = LinearRegression(fit_intercept=True)
model
```

3. Arrange data into a features matrix and target vector

- `y` is in the correct form
- Make `X` a matrix of size `[n_samples, n_features]`

```
X = x[:, np.newaxis]
X.shape
```

4. Fit the model to the data

Apply our model to the data.

- the `fit` method of the model:

$$y = X\beta + intercept$$

```
model.fit(X, y)
```

The results of these computations are stored in model-specific attributes

```
model.coef_
```

```
model.intercept_
```

5. Predict labels for unknown data

New data.

```
xfit = np.linspace(-1, 11)
```

Predict

```
Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

Supervised Learning Example: Iris Classification

Our question:

- given a model trained on a portion of the Iris data
- how well can we predict the remaining labels?

Split the data into a *training set* and a *testing set*

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
                                                random_state=1)
```

`random_state` : the seed used by the random number generator

Steps: using *Gaussian naive Bayes Classification*

```
from sklearn.naive_bayes import GaussianNB # 1. choose model class
model = GaussianNB()                       # 2. instantiate model
model.fit(Xtrain, ytrain)                  # 3. fit model to data
y_model = model.predict(Xtest)             # 4. predict on new data
```

Use the `accuracy_score` utility to see the fraction of predicted labels that match their true values:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

Unsupervised Learning Example: Iris Dimensionality

Reducing the dimensionality of the Iris data so as to more easily visualize it.

- Recall that the Iris data is four-dimensional: there are four features recorded for each sample.
- **principal component analysis** (PCA)

Steps

```
from sklearn.decomposition import PCA  # 1. Choose the model class
model = PCA(n_components=2)            # 2. Instantiate the model
model.fit(X_iris)                      # 3. Fit to data
X_2D = model.transform(X_iris)         # 4. Transform the data
```

- `n_componets` : Number of components to keep
- Species not involved!

let’s plot the results

```
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
sns.lmplot(x="PCA1", y="PCA2", hue='species', data=iris, fit_reg=False);
```

In the two-dimensional representation

- the species are fairly well separated,
- even though the PCA algorithm had no knowledge of the species labels!

Unsupervised Learning Example: Iris Clustering

Clustering: find distinct groups of data *without* reference to any labels.

- Using *Gaussian mixture model* (GMM)
- Not to be confused with *Generalized method of moments*

```
from sklearn.mixture import GaussianMixture      # 1. Choose the model class
model = GaussianMixture(n_components=3,
                        covariance_type='full')  # 2. Instantiate the model
model.fit(X_iris)                                # 3. Fit to data
y_gmm = model.predict(X_iris)                    # 4. Determine labels
```

Add the cluster label to the Iris `DataFrame` and use Seaborn to plot the results

```
iris['cluster'] = y_gmm
sns.lmplot(x="PCA1", y="PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```

Application: Exploring Handwritten Digits

Data

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
```

digits.csv

The images data is a three-dimensional array: 1,797 samples each consisting of an 8 × 8 grid of pixels.

Visualize the first hundred of these

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```

Work with this data within Scikit-Learn

- we need a two-dimensional, `[n_samples, n_features]` representation.
- flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit.
- built into the digits dataset under the `data` and `target`

```
X = digits.data
X.shape
```

```
y = digits.target
y.shape
```

Classification on Digits

Split the data into training and testing sets and fit a Gaussian naive Bayes model:

```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
```

Train

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(Xtrain, ytrain)
y_model = model.predict(Xtest)
```

Accuracy

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

Where we've gone wrong

- plot with Seaborn

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, y_model)

sns.heatmap(mat, square=True, annot=True, cbar=False, cmap='Blues')
plt.xlabel('predicted value')
plt.ylabel('true value');
```

Hyperparameters and Model Validation

The choice of model and choice of hyperparameters—are perhaps the most important part

Thinking About Model Validation

Iris dataset for example

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

Use a *k*-nearest neighbors classifier with `n_neighbors=1`

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
```

Model Validation the Right Way: Holdout Sets

Train and test split

```
from sklearn.model_selection import train_test_split
# split the data with 50% in each set
X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                  train_size=0.5)

# fit the model on one set of data
model.fit(X1, y1)

# evaluate the model on the second set of data
y2_model = model.predict(X2)
accuracy_score(y2, y2_model)
```

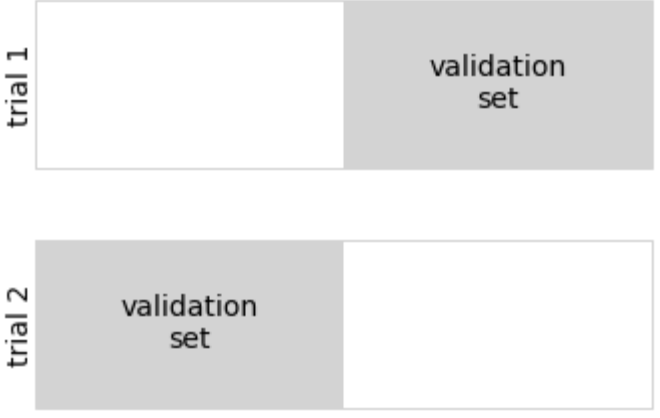
Model Validation via Cross-Validation

One disadvantage of using a holdout set for model validation

- Lost a portion of our data to the model training

Cross-validation

- to do a sequence of fits where each subset of the data is used both as a training set and as a validation set.


trial 1
validation
set
trial 2
validation
set

Demo

```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
```

Two accuracy scores:

- *two-fold cross-validation*
- Combine (by, say, taking the mean) to get a better measure of the global model performance

Scikit-Learn's `cross_val_score` convenience routine

```
from sklearn.model_selection import cross_val_score
cross_val_score(model, X, y, cv=5)
```

- Change `cv`

Leave-one-out cross validation

```
from sklearn.model_selection import LeaveOneOut
scores = cross_val_score(model, X, y, cv=LeaveOneOut())
scores
```

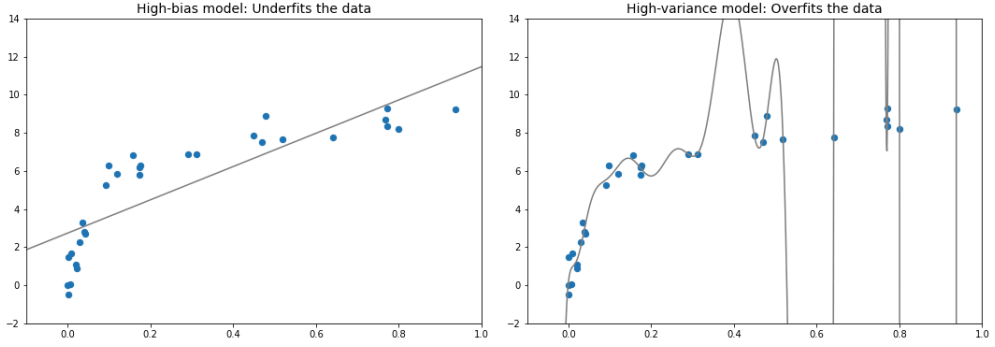
- 150 trials

Taking the mean of these gives an estimate of the error rate:

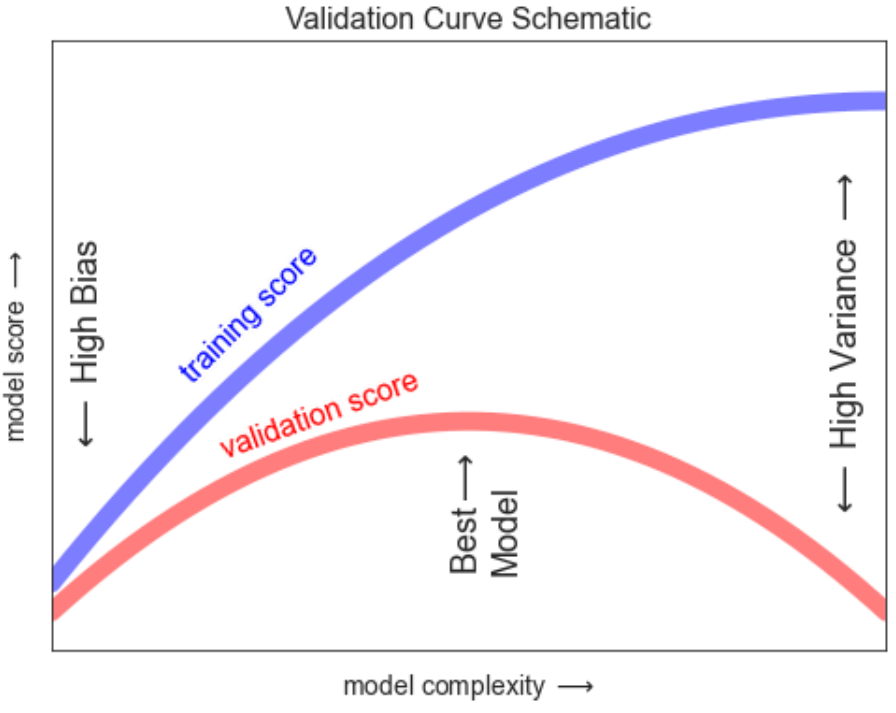
```
scores.mean()
```

Selecting the Best Model

The Bias-Variance Trade-off



The diagram shown here is often called a *validation curve*



Validation Curves in Scikit-Learn

An example of using cross-validation to compute the validation curve for a class of models.

- *polynomial regression* model

For example, a degree-1 polynomial

$$y = ax + b$$

A degree-3 polynomial fits a cubic curve

$$y = ax^3 + bx^2 + cx + d$$

A linear regression classifier combined with the polynomial preprocessor.

- use a *pipeline* to string these operations together

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
                         LinearRegression(**kwargs))
```

Create some data to which we will fit our model

```
import numpy as np

def make_data(N, err=1.0, rseed=1):
    # randomly sample the data
    rng = np.random.RandomState(rseed)
    X = rng.rand(N, 1) ** 2
    y = 10 - 1. / (X.ravel() + 0.1)
    if err > 0:
        y += err * rng.randn(N)
    return X, y

X, y = make_data(40)
```

- `.ravel()`: A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Visualize our data, along with polynomial fits of several degrees

```
import matplotlib.pyplot as plt

X_test = np.linspace(-0.1, 1.1, 500)[:, None]

plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()
for degree in [1, 3, 5]:
    y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
    plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best');
```

Automatically compute both the training score and the validation score

```
from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(
    PolynomialRegression(), X, y,
    param_name='polynomialfeatures__degree',
    param_range=degree, cv=7)

plt.plot(degree, np.median(train_score, 1),
         color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1),
         color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

? -order polynomial is the optimal

Learning Curves

Model complexity depends on the size of training data.

- A new dataset with five times as many points

```
X2, y2 = make_data(200)
plt.scatter(X2.ravel(), y2);
```

Demo

```
degree = np.arange(21)
train_score2, val_score2 = validation_curve(
    PolynomialRegression(), X2, y2,
    param_name='polynomialfeatures__degree',
    param_range=degree, cv=7)

plt.plot(degree, np.median(train_score2, 1),
         color='blue', label='training score')
plt.plot(degree, np.median(val_score2, 1),
         color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1),
         color='blue', alpha=0.3, linestyle='dashed')
plt.plot(degree, np.median(val_score, 1),
         color='red', alpha=0.3, linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

The larger dataset can support a much more complicated model:

- The peak here is probably around a degree of 6

A plot of the training/validation score with respect to the size of the training set is sometimes known as a learning curve

Learning Curves in Scikit-Learn

Compute a learning curve for our original dataset with a second-order polynomial model and a ninth-order polynomial

```
from sklearn.model_selection import learning_curve

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve(
        PolynomialRegression(degree), X, y, cv=7,
        train_sizes=np.linspace(0.3, 1, 25))

    ax[i].plot(N, np.mean(train_lc, 1),
               color='blue', label='training score')
    ax[i].plot(N, np.mean(val_lc, 1),
               color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0],
                 N[-1], color='gray', linestyle='dashed')

    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {0}'.format(degree), size=14)
    ax[i].legend(loc='best')
```

- When the learning curve has already converged *adding more training data will not significantly improve the fit!*
- The only way to *increase the converged score* is to use a different (usually more complicated) model.
- At the expense of higher model variance (indicated by the difference between the training and validation scores).

Validation in Practice: Grid Search

The trade-off between bias and variance, and its dependence on model complexity and training set size.

Scikit-Learn's `GridSearchCV` meta-estimator

Demo

```
from sklearn.model_selection import GridSearchCV

param_grid = {'polynomialfeatures__degree': np.arange(21),
              'linearregression__fit_intercept': [True, False]}

grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

Fit the model at each grid point

```
grid.fit(X, y);
```

Ask for the best parameters as follows

```
grid.best_params_
```

Use the best model and show the fit

```
model = grid.best_estimator_

plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = model.fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test);
plt.axis(lim);
```

Feature Engineering

Build feature matrix

- Categorical data
- Text

Categorical Features

Data on housing prices

- "price" and "rooms," you also have "neighborhood" information

```
data = [
    {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
    {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
    {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
    {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
]
```

How to encode "neighborhood"?

You might be *tempted* to encode this data with a **straightforward numerical mapping**:

- 'Queen Anne': 1
- 'Fremont': 2
- 'Wallingford': 3

Imply that *Queen Anne < Fremont < Wallingford*, or even that *Wallingford–Queen Anne = Fremont*

Creates extra columns indicating the presence or absence of a category with a value of 1 or 0

- Scikit-Learn's `DictVectorizer`

```
from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer(sparse=False, dtype=int)
vec.fit_transform(data)
```

- the `neighborhood` column has been expanded into three separate columns
- each row has a 1 in the column associated with its neighborhood.

To see the meaning of each column, you can inspect the feature names:

```
vec.get_feature_names_out()
```

Because the encoded data contains mostly zeros, a sparse output can be a very efficient solution:

```
vec = DictVectorizer(sparse=True, dtype=int)
vec.fit_transform(data)
```

Text Features

For example,

- most automatic mining of social media data relies on some form of encoding the text as numbers.

One of the simplest methods of encoding

- *word counts*

For example, consider the following set of three phrases:

```
sample = ['problem of evil',
          'evil queen',
          'horizon problem']
```

Construct individual columns

- representing the words "problem," "of," "evil," and so on
- using Scikit-Learn's `CountVectorizer`

Demo

```
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X = vec.fit_transform(sample)
X
```

Convert this to a `DataFrame` with labeled columns:

```
import pandas as pd
pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
```

Some issues with using a simple raw word count

- put too much weight on words that appear very frequently

Fix

- term frequency–inverse document frequency (*TF–IDF*)
- $\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$

Demo

```
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer()
X = vec.fit_transform(sample)
pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
```

or without normalization

```
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer(norm=None)
X = vec.fit_transform(sample)
pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
```

$$1.287682 \approx 1 \times \left[\log\left(\frac{1+3}{1+2}\right) + 1\right]$$

Image Features

The simplest approach is what we used for the digits data

- using the pixel values themselves

Derived Features

Mathematically derived from some input features

- For example: add polynomial features to the data

Demo

```
# data
import numpy as np
import matplotlib.pyplot as plt

x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y);
```

Fit a line to the data

```
from sklearn.linear_model import LinearRegression
X = x[:, np.newaxis]
model = LinearRegression().fit(X, y)
yfit = model.predict(X)
plt.scatter(x, y)
plt.plot(x, yfit);
```

Need a more sophisticated model to describe the relationship between x and y

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=False)
X2 = poly.fit_transform(X)
print(X2)
```

- The derived feature matrix has one column representing x,
- a second column representing x^2,
- and a third column representing x^3.

We get a closer fit

```
model = LinearRegression().fit(X2, y)
yfit = model.predict(X2)
plt.scatter(x, y)
plt.plot(x, yfit);
```

Imputation of Missing Data

Another common need in feature engineering is handling of missing data.

- `NaN` is often is used to mark missing value

Demo

```
# data
from numpy import nan
X = np.array([[ nan, 0,   3  ],
              [ 3,   7,   9  ],
              [ 3,   5,   2  ],
              [ 4,   nan, 6  ],
              [ 8,   8,   1  ]])
y = np.array([14, 16, -1,  8, -5])
```

First, imputation

- replace the missing values with some appropriate fill value
- basic approach
  - using the mean, median, or most frequent value

Scikit-Learn provides the `SimpleImputer` class:

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy='mean')
X2 = imp.fit_transform(X)
X2
```

Then fit

```
model = LinearRegression().fit(X2, y)
model.predict(X2)
```

Feature Pipelines

String together multiple steps.

For example,

1. Impute missing values using the mean.
2. Transform features to quadratic.
3. Fit a linear regression model.

Scikit-Learn provides a `Pipeline` object

Demo

```
from sklearn.pipeline import make_pipeline

model = make_pipeline(SimpleImputer(strategy='mean'),
                      PolynomialFeatures(degree=2),
                      LinearRegression())
```

This pipeline looks and acts like a standard Scikit-Learn object

```
model.fit(X, y)  # X with missing values, from above
print(y)
print(model.predict(X))
```

(Not) In Depth: Naive Bayes Classification

Example: Classifying Text

Classifying news text to topics

Let’s download the data and take a look at the target names:

```
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups()
data.target_names
```

If it fails to download, try this

- First, download file 20news-bydate_py3.pkz to your working directory `%pwd`
- Then run this instead

```
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups(data_home=".")
data.target_names
```

For simplicity here, we will select just a few of these categories and download the training and testing sets:

```
categories = ['talk.religion.misc', 'soc.religion.christian',
              'sci.space', 'comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)
```

Here is a representative entry from the data:

```
print(train.data[5][48:])
```

In order to use this data for machine learning

- need to convert the content of each string into a vector of numbers.
- use the TF-IDF vectorizer (introduced in Feature Enginneering)
- and create a pipeline that attaches it to a multinomial naive Bayes classifier

Demo

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

Apply the model to the training data and predict labels for the test data:

```
model.fit(train.data, train.target)
labels = model.predict(test.data)
```

Evaluate them to learn about the performance of the estimator

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(test.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=train.target_names, yticklabels=train.target_names,
            cmap='Blues')
plt.xlabel('true label')
plt.ylabel('predicted label');
```

We now have the tools to determine the category for *any* string

- using the `predict` method of this pipeline.

```
def predict_category(s, train=train, model=model):
    pred = model.predict([s])
    return train.target_names[pred[0]]
```

- Try it out:

```
predict_category('sending a payload to the ISS')
predict_category('discussing the existence of God')
predict_category('determining the screen resolution')
```

(Not) In Depth: Linear Regression

Example: Predicting Bicycle Traffic

Predict the number of bicycle trips based on

- weather, season, and other factors

Download data

- Bike data: FremontBridge.csv
- Weather data: SeattleWeather.csv

Use pandas to read csv files

```
import pandas as pd
counts = pd.read_csv('FremontBridge.csv',
                     index_col='Date', parse_dates=True)
weather = pd.read_csv('SeattleWeather.csv',
                      index_col='DATE', parse_dates=True)
```

Data clean

- For simplicity, look at data prior to 2020 in order to avoid the effects of the COVID-19 pandemic

```
counts = counts[counts.index < "2020-01-01"]
weather = weather[weather.index < "2020-01-01"]
```

- Compute the total daily bicycle traffic, and put this in its own `DataFrame`:

```
daily = counts.resample('d').sum()
daily['Total'] = daily.sum(axis=1)
daily = daily[['Total']] # remove other columns
```

- Adding binary columns that indicate the day of the week:

```
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
for i in range(7):
    daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

- Similarly, add an indicator of holidays

```
from pandas.tseries.holiday import USFederalHolidayCalendar
cal = USFederalHolidayCalendar()
holidays = cal.holidays('2012', '2020')
daily = daily.join(pd.Series(1, index=holidays, name='holiday'))
daily['holiday'].fillna(0, inplace=True)
```

- Add the hours of daylight

```
def hours_of_daylight(date, axis=23.44, latitude=47.61):
    """Compute the hours of daylight for the given date"""
    days = (date - pd.datetime(2000, 12, 21)).days
    m = (1. - np.tan(np.radians(latitude))
         * np.tan(np.radians(axis) * np.cos(days * 2 * np.pi / 365.25)))
    return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.

daily['daylight_hrs'] = list(map(hours_of_daylight, daily.index))
daily[['daylight_hrs']].plot()
plt.ylim(8, 17)
```

- Add the average temperature and total precipitation to the data.
- Add a flag that indicates whether a day is dry (has zero precipitation):

```
weather['Temp (F)'] = 0.5 * (weather['TMIN'] + weather['TMAX'])
weather['Rainfall (in)'] = weather['PRCP']
weather['dry day'] = (weather['PRCP'] == 0).astype(int)

daily = daily.join(weather[['Rainfall (in)', 'Temp (F)', 'dry day']])
```

Finally, add a counter that increases from day 1, and measures how many years have passed.

```
daily['annual'] = (daily.index - daily.index[0]).days / 365.
```

Look at the data

```
daily.head()
```

Fit a linear regression model

```
# Drop any rows with null values
daily.dropna(axis=0, how='any', inplace=True)

column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun',
                'holiday', 'daylight_hrs', 'Rainfall (in)',
                'dry day', 'Temp (F)', 'annual']
X = daily[column_names]
y = daily['Total']

model = LinearRegression(fit_intercept=False)
model.fit(X, y)
daily['predicted'] = model.predict(X)
```

Compare the total and predicted bicycle traffic visually (see the following figure):

```
daily[['Total', 'predicted']].plot(alpha=0.5);
```

The data and model predictions don't line up exactly,

- Nevertheless, our rough approximation is enough

How much each feature contributes to the daily bicycle count:

```
params = pd.Series(model.coef_, index=X.columns)
params
```

- There are thousands fewer riders on weekends than on weekdays.
- Skipped: compute these uncertainties quickly using bootstrap resamplings of the data

(Not So) In Depth: Support Vector Machines

Supervised algorithms for classification

Motivating Support Vector Machines

discriminative classification

Find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

For example

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

A linear discriminative classifier would attempt to draw a straight line separating the two sets of data

Problem: there is more than one possible dividing line

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
    plt.plot(xfit, m * xfit + b, '-k')

plt.xlim(-1, 3.5);
```

Support Vector Machines: Maximizing the Margin

margin

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                     color='lightgray', alpha=0.5)

plt.xlim(-1, 3.5);
```

The line that maximizes this margin is the one we will choose as the optimal model.

Fitting a Support Vector Machine

use Scikit-Learn's support vector classifier (`SVC`) to train an SVM model

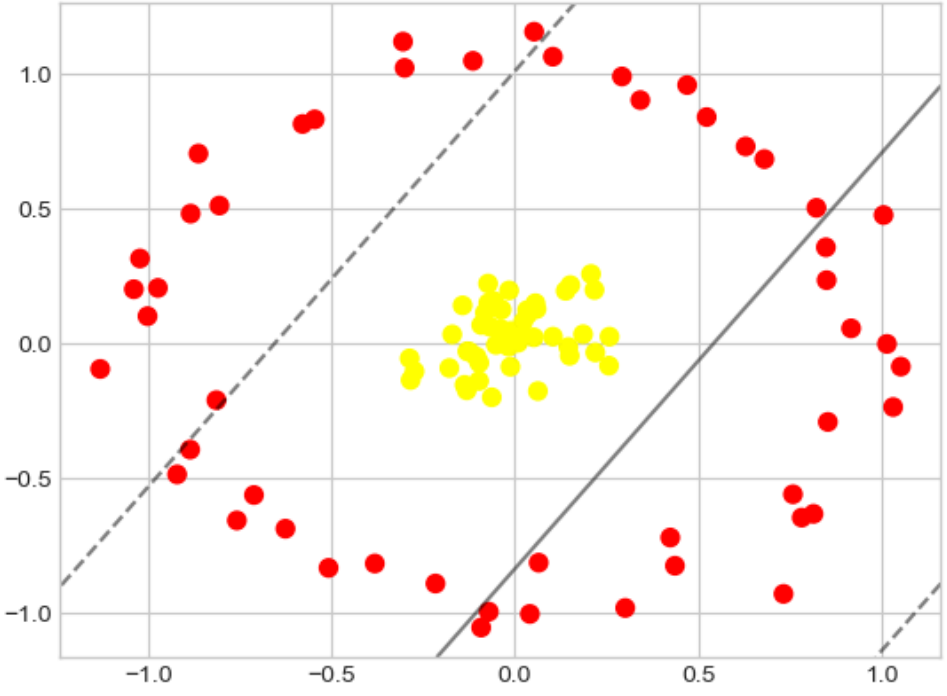
```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10) # linear kernel and a very large C
model.fit(X, y)
```

```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, edgecolors='black',
                   facecolors='none');
```

Plot boundaries

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(model);
```

Beyond Linear Boundaries: Kernel SVM



Not linearly separable

radial basis function 径向基核函数 (RBF)

```
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

Visualize the fit and identify the support vectors

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
            s=300, lw=1, facecolors='none');
```

Tuning the SVM: Softening Margins

When data has some amount of overlap?

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=1.2)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

"softens" the margin:

- it allows some of the points to creep into the margin if that allows a better fit.
  - For a very large `C`, the margin is hard, and points cannot lie in it.
  - For a smaller `C`, the margin is softer

Demo

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
    axi.set_title('C = {0:.1f}'.format(C), size=14)
```

Example: Face Recognition

Face to label

Download the faces datasets

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

If it fails, try this

- Download lfw_home.zip
- Extracted to `%pwd`
- Then run

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(data_home=".", min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

Plot a few of these faces to see what we're working with

```
fig, ax = plt.subplots(3, 5, figsize=(8, 6))
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[],
            xlabel=faces.target_names[faces.target[i]])
```

Each image contains 62 × 47, or around 3,000, pixels.

- Use *principal component analysis* to extract 150 fundamental components

The model

```
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline

pca = PCA(n_components=150, whiten=True,
          svd_solver='randomized', random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')
model = make_pipeline(pca, svc)
```

Train and test

- split the data into a training set and a testing set:

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data, faces.target,
                                                random_state=42)
```

- use grid search cross-validation to explore combinations of parameters

```
from sklearn.model_selection import GridSearchCV
param_grid = {'svc__C': [1, 5, 10, 50],
              'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
grid = GridSearchCV(model, param_grid)

%time grid.fit(Xtrain, ytrain)
print(grid.best_params_)
```

The optimal values fall toward the middle of our grid;

- if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

Predict the labels for the test data

```
model = grid.best_estimator_
yfit = model.predict(Xtest)
```

Plot test images along with their predicted values

```
fig, ax = plt.subplots(4, 6)
for i, axi in enumerate(ax.flat):
    axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
    axi.set(xticks=[], yticks=[])
    axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                   color='black' if yfit[i] == ytest[i] else 'red')
fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```

Get estimator's performance using the classification report

```
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit,
                            target_names=faces.target_names))
```

Display the confusion matrix between these classes

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
mat = confusion_matrix(ytest, yfit)
sns.heatmap(mat.T, square=True, annot=True, fmt='d',
            cbar=False, cmap='Blues',
            xticklabels=faces.target_names,
            yticklabels=faces.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```

END