Simone Scardapane {simone.scardapane@uniroma1.it}

A practical introduction to machine learning with scikit-learn
What is machine learning?
"Machine learning explores the study and construction of algorithms that can **learn** from and make **predictions** on data. Such algorithms operate by building a **model** from example inputs in order to make data-driven predictions or decisions."

https://en.wikipedia.org/wiki/Machine_learning
Why machine learning?

A survey of 168 enterprises with at least $500M in sales report that:

1. **76%** are targeting **higher sales growth** with machine learning.

2. At least **40%** are already using **machine learning** to improve sales and marketing performance.

1. More data coming from everywhere (Big Data…)
2. More computing power (Hadoop, Spark…)
3. Stable and *simple* software libraries for standard problems
Machine learning problems

**Supervised learning**
- **Classification**: learn to classify an object
- **Regression**: just like classification, with real outputs

**Recommending**

**Unsupervised learning**
- **Clustering**: objects into groups
- **Dimensionality reduction / compression**

**Reinforcement learning**
Machine learning in Python
Full stack solutions: Anaconda; WinPython; Canopy
Example 1
Music classification
Workflow

1. Define problem
2. Collect data
3. Extract features
4. Preprocessing
5. Train classifier
6. Deploy model
Task: classify a song into a musical genre *automatically.*

https://screenshots.en.sftcdn.net/en/scrn/319000/319672/pandora-radio-17-700x437.png
Workflow

Define problem
Collect data
Extract features
Train classifier
Preprocessing
Deploy model
Dataset: **Garageband**, a set of 1856 songs ("examples") **manually** labeled according to 9 genres (e.g., pop, rock…).

Input: 49 values ("features") representing cepstral information on each song.

Steps we do not consider explicitly today:

- Feature extraction (depends on the input data)
- Handling missing data
Step 1 - Load the data

```python
# Step 1: we load the audio
import scipy.io
data = scipy.io.loadmat('example_1_music.mat')
```

**Song representation**

```
>> In [25]: data['songs'][0]
array([ 5.1993087e-02, 7.80682612e-02, 3.82337187e-02, ... ,2.21258000e+05])
```

**Desired label**

```
>> In [26]: data['labels'][0]
'blues'
```

**Song name**

```
>> In [27]: data['names'][0]
'Los_Blancos_Shakin__That_Thing_mp3'
```
Workflow

1. Define problem
2. Collect data
3. Extract features
4. Preprocessing
5. Train classifier
6. Deploy model
All objects in sklearn have two methods:

- **fit** to train the object on the available data
- **predict** to apply a supervised object to data
- **transform** to apply an unsupervised object to data

The function **score** can be used to obtain a default accuracy metric (task dependent).
Step 2 - Train the model

```python
import sklearn.svm

svc = sklearn.svm.LinearSVC(C=0.01)
svc.fit(data['songs'], data['labels'])

# We verify the accuracy (training)
score_1 = svc.score(data['songs'], data['labels'])
```

score_1 = 0.26 → 26% probability of being correct on the training data (poor!)
Feature normalization

As a general rule, all the inputs of a model should be on the same scale.

Consider the following:

<table>
<thead>
<tr>
<th>Student</th>
<th>Age (years)</th>
<th>Income (€/month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>18</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>19</td>
<td>5,000</td>
</tr>
<tr>
<td>C</td>
<td>60</td>
<td>600</td>
</tr>
</tbody>
</table>

Euclidean distance ("similarity") $A/B \approx 4,500$
Euclidean distance $A/C \approx 108$
Step 3 – Normalize the features

MinMaxScaler can be used to transform all features in a predefined range (by default [0,1]).

```python
import sklearn.preprocessing
scaler = sklearn.preprocessing.MinMaxScaler()
data['songs'] = scaler.fit_transform(data['songs'])

svc.fit(data['songs'], data['labels'])
score_2 = svc.score(data['songs'], data['labels'])
```

Accuracy has improved to 52.80%... But is it reliable?
Overfitting

The training error of the green curve is 0%!

Intuitively, everyone chooses the black curve…
You want to model the informative part, and discard noise.

However, a flexible model can adapt to both components.

Conversely, a very simple model is not flexible enough.

All supervised algorithms have a way to control this trade-off.
Step 4 – Train / test split

A simple solution: keep an independent *test set* to evaluate the model.

```python
# Train / test split
from sklearn.cross_validation import train_test_split
(songs_trn, songs_tst) = train_test_split(data['songs'], test_size=0.33, random_state=1)
(labels_trn, labels_tst) = train_test_split(data['labels'], test_size=0.33, random_state=1)

# Train and test on separate splits
svc.fit(songs_trn, labels_trn)
score_3 = svc.score(songs_tst, labels_tst)
```

The new score is only slightly worse, so the model seems reliable enough.
We can do better with k-fold cross-validation

http://blog.kaggle.com/2015/06/29/scikit-learn-video-7-optimizing-your-model-with-cross-validation/
Nested test/validation

http://sebastianraschka.com/faq/docs/evaluate-a-model.html
Step 5 – Hyper-parameter optimization

By optimizing the internal parameter, the accuracy has improved to 60%. Is it good?

```python
# Optimize the parameter of the model
import sklearn.grid_search
import numpy as np
params = {'C': np.power(2.0, np.arange(-10, 1))}
svc_optimized = sklearn.grid_search.GridSearchCV(svc, param_grid=params, verbose=2)

svc_optimized.fit(songs_trn, labels_trn)
score_4 = svc_optimized.score(songs_tst, labels_tst)
```
How good is a model?

A model is good \textit{with respect to} something else – difficulty depends on the problem.

Check against a very simple model e.g., predicting the output uniformly at random.

In this case, "dummy" accuracy is 12\% - problem is difficult!

```python
# Compare to a dummy classifier
import sklearn.dummy
dummy = sklearn.dummy.DummyClassifier(strategy='uniform')
score_dummy = dummy.fit(songs_trn, labels_trn).score(songs_tst, labels_tst)
```
Analyzing the confusion matrix
Example 2
Image segmentation
What is clustering?

Clustering is the problem of partitioning elements into a number of groups, so that:

1. Points in the same cluster are "similar"
2. Points in different clusters are "dissimilar"
A single pixel is described by \((x,y)\) position and intensity.

**Cluster**: a group of *nearby* pixels with similar color.

*Roughly* similar to image segmentation.
Load the image (simplified)

```python
# Load image from PIL
import Image
import numpy as np
im = Image.open("photo.jpg")
im = np.array(im)

# Construct dataset for clustering
d = list()
for i in range(im.shape[0]):
    for j in range(im.shape[1]):
        d.append((i,j,im[i,j]))
d = np.array(d).astype(np.float64)

Final data matrix is $N \times 3$, where $N$ is the number of pixels in the original image.
```
Normalize and cluster

# Normalize
import sklearn.preprocessing
p = sklearn.preprocessing.MinMaxScaler()
d = p.fit_transform(d)

# Cluster with 5 clusters
import sklearn.cluster
c = sklearn.cluster.KMeans(n_clusters=5)
c.fit(d)

# Show the resulting image
import matplotlib.pyplot as plt
plt.imshow(c.labels_.reshape(im.shape))
Visualizing the results

\( K = 2 \)  \( K = 4 \)  \( K = 7 \)
Example 3
Joke recommendation
A recommender system gives personalized recommendations to users ("you may be interested in…”).

Collaborative filtering solves this by analyzing preferences of "similar" users:

1. $N$ users.
2. $P$ products.
3. Each user has rated some products.
4. You want to infer all the other ratings or an interesting subset.
Collaborative filtering (example)

https://commons.wikimedia.org/wiki/File:Collaborative_filtering.gif
Examples of tasks

Recommendation and Ratings Public Data Sets For Machine Learning

Movies Recommendation:

- MovieLens - Movie Recommendation Data Sets http://www.grouplens.org/node/73
- Jester - Movie Ratings Data Sets (Collaborative Filtering Dataset) http://www.ier.berkeley.edu/~goldberg/jester-data/
- Cornell University - Movie-review data for use in sentiment-analysis experiments http://www.cs.cornell.edu/people/pabo/movie-review-data/

Music Recommendation:

- Amazon - Audio CD recommendations http://131.193.40.52/data/

Books Recommendation:

- Institut für Informatik, Universität Freiburg - Book Ratings Data Sets http://www.informatik.uni-freiburg.de/~cziegler/BX/

Food Recommendation:

https://gist.github.com/entaroadun/1653794
"Over 4.1 million continuous ratings (-10.00 to +10.00) of 100 jokes from 73,421 users: collected between April 1999 - May 2003."

http://eigentaste.berkeley.edu/dataset/
Recommending in Python

Python has small support for recommending; some options:

- **MLib** in Spark.
- **Crab** (scikits add-on).
- **Nimfa**.
- **Graphlab Create**.

Here, we use a decomposition technique of sklearn, although *this is not optimal.*
Matrix factorization (hint)

Each rating results from the (linear) interaction of $r$ factors describing a user and $r$ factors describing a joke.

We can use the factors to estimate the missing values in the original ratings' matrix,
The code

```
# Load the data
import pandas as pd
d = pd.read_excel('jester-data-1.xls').values[:,1:]

# Normalize ratings
d = (d + 10)/20
d[d == 5.450] = 0 # Execute factorization
import sklearn.decomposition.nmf
nmf = sklearn.decomposition.nmf.NMF(n_components=10)
d = nmf.fit_transform(d)
```

Only $\approx 37,500$ missing values instead of $687,819$. 
Where to from here?

Follow a **MOOC** course: [Introduction to Machine Learning (Udacity)](https://.udacity.com/

Follow (another) **MOOC** course: [Machine Learning (Coursera)](https://www.coursera.org/

Try some cloud services: [Machine Learning Cloud Service (Google)](https://cloud.google.com/ml-engine)

**Participate in machine learning competitions (Kaggle)!**


Experiment, experiment, experiment…
A practical introduction to machine learning with scikit-learn

Simone Scardapane
GDG L-ab Member

Post-doc research fellow @ La Sapienza
simone.scardapane@uniroma1.it

< Grazie dell’attenzione! >