Python programming — machine learning

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

October 28, 2013
Overview

Machine learning: Unsupervised and supervised

The concept of generalization

Overview of Python machine learning packages

Computations in a naïve Bayes classifier

Example with Pima data set: Baseline, linear, random forest.
Machine learning

Unsupervised learning

You have multiple data items each with (usually) multiple features

Example: Topic mining

Supervised learning

You have multiple data items each with (usually) multiple features and an associated label (or continuous values) to each data item.

You usually want to predict unknown labels

Example: spam detection, sentiment analysis
Supervised learning

Regression (usually when labels and classification are continuous)

“Classical statistics”: Linear regression, multiple regression.

(Artificial) neural networks: Nonlinear model

Naïve Bayes Classifier: A model with assumption of independence between features.

Others: “Support vector machine”, “Random forest”, …
Generalization

Generalization = How the (trained) model perform on new unseen data.

When you estimate/train a machine learning model on (training) data you will likely “overfit”, i.e., the model parameters are fitted to the data “too much”.

To get a unbiased estimate of the performance of the machine learning model you need to split the data and train on one part and test on the other.

Important concept: Cardinal sin to evaluate the performance on the training set.
## Python machine learning package

<table>
<thead>
<tr>
<th>Name</th>
<th>KLoC</th>
<th>GS-cites</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scipy.linalg</td>
<td>92</td>
<td>2</td>
<td>(Seabold and Perktold, 2010)</td>
</tr>
<tr>
<td>Statsmodels</td>
<td>398</td>
<td>278</td>
<td>(Pedregosa et al., 2011)</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>286</td>
<td>0 (yet!)</td>
<td>(Demšar et al., 2013)</td>
</tr>
<tr>
<td>Orange</td>
<td>75</td>
<td>8</td>
<td>(Albanese et al., 2012)</td>
</tr>
<tr>
<td>mlpy</td>
<td>31</td>
<td>58</td>
<td>(Zito et al., 2008)</td>
</tr>
<tr>
<td>MDP</td>
<td>9</td>
<td>62</td>
<td>(Řehůřek and Sojka, 2010)</td>
</tr>
<tr>
<td>Gensim</td>
<td>214</td>
<td>527</td>
<td>(Bird et al., 2009)</td>
</tr>
<tr>
<td>NLTK</td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Python supervised learning

Linear regression: `scipy.linalg.lstsq`

**scikit-learn**: Lots of algorithms for supervised learning

NLTK has its own naïve Bayes classifier (`nltk.NaiveBayesClassifier`) and access to external classifiers, e.g., **scikit-learn** (if setup)
Naïve Bayes classifier (NBC)

Described in Python books with full implementation (Segaran, 2007, pp. 123–127, 277–281) or just the functional interface to the NTLK implementation (Bird et al., 2009, chapter 6)

“Bayes” because the Bayes theorem is used on the features/labels:

\[
P(label|features) = \frac{P(features|label) P(label)}{P(features)}
\]  

(1)

“Naïve” because of a strong (usually wrong) assumption about independence of the features:

\[
P(features|label) = \prod_{n} P(feature_n|label)
\]  

(2)

Note that the features in the NBC might be both categorical (binary, multiple classes) or continuous, but that the Python books present the NBC for binary features.
Example with NLTK

Small data set with two documents, one spam and one “ok”:

Buy Viagra
Hello dear

NLTK NBC trained with features (here: words) extracted from the data set and then classifying a new document:

```
import nltk
labeled_featuresets = [({'viagra': True, 'hello': False}, 'spam'),
                       ({'viagra': False, 'hello': True}, 'ok')]
classifier = nltk.NaiveBayesClassifier.train(labeled_featuresets)
classifier.prob_classify({'viagra': True, 'hello': False}).prob('spam')
0.9
```

So what does “0.9” mean? An estimate of the probability that the new document is spam!
Contingency table of documents

NBC makes estimation from data represented in a contingency table:

<table>
<thead>
<tr>
<th>Features/Labels</th>
<th>Spam</th>
<th>Ok</th>
<th>...</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viagra (True)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>¬ Viagra (False)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table body:

```python
>>> [(k,v.freqdist().items()) for k,v in
classifier._feature_probdist.items()]
[('ok', 'hello'), [(True, 1)], ('ok', 'viagra'), [(False, 1)],
 ('spam', 'hello'), [(False, 1)], ('spam', 'viagra'), [(True, 1)]
]
```

Last row:

```python
>>> [(k,v) for k,v in classifier._label_probdist.freqdist().items()]
[('ok', 1), ('spam', 1)]
```
Naïve naïve Bayes classifier

“Naïve naïve Bayes classifier”: estimation of probabilities with maximum likelihood:

\[
P(\text{viagra}|\text{spam}) = \frac{P(\text{viagra, spam})}{P(\text{spam})} \approx \frac{\text{count(\text{viagra, spam})}}{\text{count(\text{spam})}} = \frac{1}{1} = 1 \quad (3)
\]

\[
P(\text{hello}|\text{spam}) = \frac{P(\text{hello, spam})}{P(\text{spam})} \approx \frac{0}{1} = 0 \quad (4)
\]

But “hello” might occur in some spam documents, so it might not be a good idea to estimate the probability as zero.

To get around that problem a technique is used that in text mining is called smoothing.
Smoothing

NLTK implements smoothing in `nltk.probability`: “Lidstone estimate”, “expected likelihood estimate” (ELE), ...

\[ P(\text{feature}|\text{label}) \approx \frac{\text{count(feature, label)} + \gamma}{\text{Total} + B\gamma}, \]  

where \( \gamma \) is the amount of smoothing and \( B \) is the number of labels.

For ELE \( \gamma = 0.5 \).

Wikipedia refers to this as “additive smoothing”.

This kind of smoothing is also closely associated with what in statistics/machine learning is called the Dirichlet prior for the multinomial distribution.
Prior

Prior probabilities is unknown, but usually estimated/set to the relative frequency the different labels occur in the full data set:

\[ P(\text{spam}) = \frac{\text{count}(\text{spam})}{\text{count}(\text{spam}) + \text{count}(\text{ok})} = \frac{1}{1 + 1} \]

This is the maximum likelihood estimate.

Default estimation in NLTK is ELE \((\gamma = 0.5)\) and the estimated probability distribution is available in `classifier._label_probdist`

Example with two spam document and one ok document the priors are computed as:

\[ P(\text{spam}) = \frac{2 + 0.5}{3 + 1} = 0.675 \]
\[ P(\text{ok}) = \frac{1 + 0.5}{3 + 1} = 0.375 \]
Probabilistic prediction computations

\[
\begin{align*}
P(\text{spam}) &= \frac{1 + 0.5}{2 + 2 	imes 0.5} = 0.5 \\
P(\text{ok}) &= \frac{1 + 0.5}{2 + 2 	imes 0.5} = 0.5 \\
P(\text{viagra}|\text{spam}) &= \frac{1 + 0.5}{1 + 1} = 0.75 \\
P(\text{viagra}|\text{ok}) &= \frac{0 + 0.5}{1 + 1} = 0.25 \\
P(\text{hello}|\text{spam}) &= \frac{0 + 0.5}{1 + 1} = 0.25 \\
P(\text{hello}|\text{ok}) &= \frac{1 + 0.5}{1 + 1} = 0.75 \\
P(\text{features}) &= \sum_{m} \prod_{n} P(\text{feature}_n|\text{label}_m) \cdot P(\text{label}_m) \\
P(\text{spam}|\text{viagra}, \neg\text{hello}) &= \frac{\prod_n P(\text{feature}_n|\text{spam}) \cdot P(\text{spam})}{P(\text{features})} \\
&= \frac{0.75 \times 0.75 \times 0.5}{0.75 \times 0.75 \times 0.5 + 0.25 \times 0.25 \times 0.5} \\
&= \frac{0.28125}{(0.28125 + 0.03125)} \\
&= 0.9
\end{align*}
\]
Another dataset

Now with three features:

```python
>>> labeled_featuresets = [
    ({'viagra': True, 'buy': True, 'hi': False}, 'spam'),
    ({'viagra': True, 'buy': False, 'hi': False}, 'spam'),
    ({'viagra': False, 'buy': False, 'hi': True}, 'ok')]
>>> classifier = nltk.NaiveBayesClassifier.train(labeled_featuresets)
>>> print(classifier._label_probdist.freqdist())
<FreqDist: 'spam': 2, 'ok': 1>
```

Prior on 0.625 equal to:

\[
\frac{2 + 0.5}{3 + 2 \times 0.5} = 0.625
\]
Most informative feature

The NLTK NBC has a convenient method to display “informative” features (most_informative_features/show_most_informative_features).

How “informative” a feature \( n \) is, is computed as separated for each feature \( n \)

\[
I_n = \max_{\text{label}} \left( \frac{P(\text{feature}_n | \text{label})}{\min_{\text{label}} (P(\text{feature}_n | \text{label}))} \right)
\]

For the “viagra” feature:

\[
I_{\text{viagra}} = 0.71/0.2 = 3.57 \\
I_{\neg\text{viagra}} = 0.6/0.14 = 4.2
\]

Note that there is a bug/feature in the present version (naivebayes.py 2063, 2004-07-17) of the most_informative_features and show_most_informative_features methods so that not all features are show for the present example. :-(
Manual computations

\[
\begin{align*}
P(\text{spam}) & = 0.675 \\
P(\text{viagra}|\text{spam}) & = \frac{(2 + 0.5)}{(2 + 1)} = 0.833 \\
P(\text{viagra}|\text{ok}) & = \frac{(0 + 0.5)}{(1 + 1)} = 0.25 \\
P(\neg\text{viagra}|\text{spam}) & = \frac{(0 + 0.5)}{(2 + 1)} = 0.166 \\
P(\text{buy}|\text{spam}) & = \frac{(1 + 0.5)}{(2 + 1)} = 0.5 \\
P(\neg\text{buy}|\text{spam}) & = \frac{(1 + 0.5)}{(2 + 1)} = 0.5 \\
P(\text{hi}|\text{spam}) & = \frac{(0 + 0.5)}{(2 + 1)} = 0.166 \\
P(\neg\text{hi}|\text{spam}) & = \frac{(2 + 0.5)}{(2 + 1)} = 0.833 \\
P(\neg\text{hi}|\text{ok}) & = \frac{(0 + 0.5)}{(1 + 1)} = 0.25 \\
P(\text{features}) & = \sum_{m} \prod_{n} P(\text{feature}_n|\text{label}_m) P(\text{label}_m) \\
P(\text{features}, \text{spam}) & = 0.833 \times 0.5 \times 0.833 \times 0.675 = 0.234375 \\
P(\text{features}, \text{ok}) & = 0.25 \times 0.75 \times 0.25 \times 0.325 = 0.015234375 \\
P(\text{spam}|\text{viagra}, \neg\text{buy}, \neg\text{hi}) & = \frac{P(\text{features})}{P(\text{features})} \\
& = \frac{0.234375}{0.234375 + 0.015234375} = 0.938967
\end{align*}
\]
Probabilistic prediction computation

```python
>>> classifier.prob_classify({'viagra': True,
                             'hi': False,
                             'buy': False}).prob('spam')
0.938213041911853
```

Notice the small difference between the manual and the NLTK results! :-o

The present version of NLTK handles feature/label combinations it hasn’t seen (e.g., viagra=True, ok) in a special way!
Pima data set machine learning
Get a data set

Get a data set consisting of training and testing parts from $R$:

$ R$
> library(MASS)
> write.csv(Pima.tr, "Pima.tr.csv")
> write.csv(Pima.te, "Pima.te.csv")

We want to predict the “type” column of each row from Its either “yes” or “no” indicate whether a subject has diabetes or not.

This data set is not sparse.
First performance measure function

```python
def accuracy(truth, predicted):
    if len(truth) != len(predicted):
        raise Exception("Wrong sizes ...")
    total = len(truth)
    if total == 0:
        return 0
    hits = len(filter(lambda (x, y): x == y, zip(truth, predicted)))
    return float(hits)/total

def test_accuracy():
    assert accuracy([], []) == 0
    assert accuracy([1], [1]) == 1.0
    assert accuracy([1], [0]) == 0.0
    assert accuracy([1, 2, 3, 4], [1, 5, 3, 7]) == 0.5
```

Finn Årup Nielsen 20 October 28, 2013
Baseline model

“Training” a baseline model:

```python
>>> import pandas as pd
>>> Pima_tr = pd.read_csv("Pima.tr.csv", index_col=0)
>>> Pima_te = pd.read_csv("Pima.te.csv", index_col=0)
>>> Pima_tr.groupby("type").count()
	npreg  glu  bp  skin  bmi  ped  age  type
    type
    No   132  132  132  132  132  132  132  132
    Yes  68   68   68   68   68   68   68   68
```

There are more “type” = “no” in the training data set.

Lets just predict “no”.
“No” classifier

class NoClassifier():
    """Classifier that predict all data as "No". """
    def predict(self, x):
        return pd.Series(["No"] * x.shape[0])

no_classifier = NoClassifier()
predicted = no_classifier.predict(Pima_te)

>>> accuracy(Pima_te.type, predicted)
0.6716867469879518

Around 67% prediction accuracy on the test set.
Linear model with scipy.linalg ...

```python
from scipy.linalg import pinv
from numpy import asarray, hstack, mat, ones, where

class LinearClassifier():
    """ y = X*b and b = pinv(X) * y """
    def __init__(self):
        self._parameters = None

    def from_labels(self, y):
        return mat(where(y=="No", -1, 1)).T

    def to_labels(self, y):
        return pd.Series(asarray(where(y<0, "No", "Yes"))).flatten()

    def train(self, x, y):
        intercept = ones((x.shape[0], 1))
        self._parameters = pinv(hstack((mat(x), intercept))) * self.from_labels(y)

    def predict(self, x):
        intercept = ones((x.shape[0], 1))
        y_estimated = hstack((mat(x), intercept)) * self._parameters
        return self.to_labels(y_estimated)
```

Finn Årup Nielsen
... **Linear model**

```python
lc = LinearClassifier()
lc.train(Pima_tr.ix[:, :7], Pima_tr.type)

predicted = lc.predict(Pima_te.ix[:, :7])

accuracy(Pima_te.type, predicted)
```

Now we get 0.7981927710843374. Somewhat better.

Note the hassle with conversion from and to labels “no” and “yes” and between matrices, arrays and Pandas Series.
Scikit-learn Random Forest

Here the data is converted to numerical arrays and vectors

```python
from numpy import where

X_tr = Pima_tr.ix[:, :7]
y_tr = where(Pima_tr.type=="No", -1, 1)
X_te = Pima_te.ix[:, :7]
y_te = where(Pima_te.type=="No", -1, 1)
```

```python
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X_tr, y_tr)
>>> predicted = where(rfc.predict(X_te)==-1, "No", "Yes")
0.7228915662650602
>>> rfc.score(X_te, y_te)
0.72289156626506024
```

Ups, not better. Note that the result may vary
ROC curve

See Receiver operating characteristic example using functions from the sklearn.metrics module

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

fpr, tpr, thresholds = roc_curve(y_te, rfc.predict_proba(X_te)[:,1])
roc_auc = auc(fpr, tpr)

plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.show()
```
Summary

There is a number of different packages for machine learning and prediction.

Orange and Scikit-learn has a very high developer activity

Note memory issues: The NLTK classifier might use a lot of memory
References


