Machine Learning

Jan Rygl

rygl@fi.muni.cz

jan.rygl@phonexia.com

Outline

1. What is machine learning
2. Common concepts
3. Techniques overview
4. Recommendations

Machine learning

Machine learning is the science of getting computers to act without being explicitly programmed. (Coursera)

The basic premise of machine learning is to build algorithms that can receive input data and use statistical analysis to predict an output value within an acceptable range. (whatis.com)

ML consists of:

1. goal definition
2. data acquisition
3. data preprocessing
4. feature extraction
5. a) applying ML model
6. b) output postprocessing (optional)
7. analysing results
8. deploying trained model

Goal definition

Problem type:

- one-class classification (one label per document),
- multi-class classification (zero to many labels per document),
- clustering (known number of clusters/unknown cluster count),
- regression (predict correct value),
- verification (compute similarity of two documents),
- on-line learning (train system with new inputs),
- reinforcement learning (update system with delays -- after inputs are classified),
- ...

Goal definition

Scoring:

- Which results are good/acceptable/bad?
- What is priority?
- Can I make mistakes? Can I not answer?

$$\text{Precision} = \frac{|TP|}{|TP \cup FP|}$$

$$\text{Recall} = \frac{|TP|}{|TP \cup FN|}$$

$$\text{F-measure} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{|TP \cup TN|}{|\text{All documents}|}$$

$$\text{Balanced accuracy} = \frac{|TP|}{2 \cdot |FN \cup TP|} + \frac{|TN|}{2 \cdot |FP \cup TN|}$$

Examples:

- Bank searching for account hijacks
  - We want high recall (find all possible frauds).
  - We want high precision: we have 10 people for checking accounts manually, we can have maximum 500 false positive cases per day.
  - Metric consists of two parts:
    - precision > 98% if 2% corresponds to 500 checks per day
    - recall highest possible
    - possible score:

$$score = \text{recall} + min(0, \text{precision} - 0.98) \cdot \text{PENALT}$$

Examples:

- Training language model
  - We want high accuracy
  - For English and German, we have many testing examples
  - For smaller languages such as Czech, we have less examples
  - We can train a model perfect for English and German and bad for other languages with **accuracy**
  - We don't care about **precision** and **recall**, we care about correct results
  - **Balanced accuracy** or **F-measure**: the most possible correct results for each language (F-measure is harder to explain to customer)

Goal definition

Computing power and time:

- Do we have GPU? Enough RAM? How many processor cores?
- Should system be parallel/serial?
- Should system be scalable?
- Is system on-line (response in miliseconds)?

Marker search (state of the art analysis) is recommended before starting any experiment. Which algorithms need big machines? Which techniques support on-line training? Etc.

Data acquisition

Beware of GDPR http://www.eugdpr.org/ (http://www.eugdpr.org/)

- Buy (legal!) data
- Collect and annotate
- Crawler from web vs pay somebody for crawling web
- Ask your partners/customers for data

For specific tasks (analysis of bank accounts), only one data source is enough (account logs from bank).

For general tasks (entity detection, many data sources will be needed (known entity collections, text similar to analysed texts).

Data preprocessing

- Deduplication
- Remove meta-data and other information connected to labels not present in real data
  - e.g., authorship attribution of anonymous e-mails trained on signed data
- Cleaning (remove noise such as images, tables, quotes, not always aplicable, depends on task):
  - topic recognition: ignore tables with numbers, convert images to keywords (title, label), keep quotes
  - authorship recognition (replace images and tables by tag IMG, remove quotes)
- Possible language analysis (tokenization, morphology and syntactic analysis, semantic analysis) and connecting with other databases

- Start with something simple in the beginning.
- Use existing tools, don't invent a wheel

In [1]:
```
from nltk.tokenize import word_tokenize

def my_cool_tokeniser(text):
    for char in '() .,-':
        text = text.replace(char, '|' + char + '|')
    return [token for token in text.split('|') if token and not to
ken.isspace()]
text = "Ernest Joyce (c. 1875-1940) was a Royal Naval seaman."
print(text.split())
print(word_tokenize(text))
print(my_cool_tokeniser(text))
```

```
['Ernest', 'Joyce', '(c.', '1875-1940)', 'was', 'a', 'Royal', '
Naval', 'seaman.']
['Ernest', 'Joyce', '(', 'c.', '1875-1940', ')', 'was', 'a', 'R
oyal', 'Naval', 'seaman', '.']
['Ernest', 'Joyce', '(', 'c', '.', '1875', '-', '1940', ')', 'w
as', 'a', 'Royal', 'Naval', 'seaman', '.']
```

Which text output is better? Which tokenizer is better?

Feature extraction

- **Iterative process**
- **Begin** with **simple** ones, add complexity with more experiments if not successful
- Think about features -- many features help only for one data source
- Features shouldn't be able to describe every document in training set perfectly: e.g. bag of words with all words for long books can match 100% training data easily.
- Features should be able to find out some **generalization** of rule.
- Use feature selection methods (e.g. entropy based) if you have too many features (depends on computing power and number of instances)

Feature extraction

Good starters:

- bag of words
- stop words
- word n-grams
- character n-grams

Always try to precompute features from data, don't use public lists of best words/stop words/character n-grams. They are topic and style dependent.

Feature extraction

- requires tuning data set
- data don't have to be annotated
- they need to have the same format and style as classified data
- not part of train or test set

If you extract features from training data, you can overtrain. -- Train data accuracy will be high, but test data won't be recognized well.

If you extract features from test data, you are cheating. -- Evaluation doesn't have any sense.

Preparing data for machine learning

Check that you have enough data.

Prepare:

- tune set (unlabeled, see feature extraction)
- train set
- test set

Train set is usually bigger than test set, but test set must be **representative**. Use different source of data for testing if possible and add some data of the same type as training set documents to it.

Not enough data

If you don't have enough data from making big enough train and test sets, use one of following techniques:

- **random sampling:**
  - repeat N times (e.g. N=10, n=50, ...):
  - select randomly 90% of instances and use them for training
  - rest of instances is used for testing
  - evaluate trained model
  - compile all evaluations (average, min, ...), system performance is compilation of performances of random data samplings
- **n-fold cross-validation:**
  - divide data into N groups (folds)
  - repeat N times (common values are 4 and 10):
    - Nth fold is testing, all other folds are merged into training set
    - evaluate trained model
  - compile all evaluations, same as previous step

Applying machine learning model

1. Select correct model
2. Use correct model correctly

Select the "lightest" possible model which is able to process data => you will save computing and programming time.

For Python programmers, I definitely recommend Sklearn: http://scikit-learn.org/ (http://scikit-learn.org/)

Lazy programmer is a good programmer.

Implementing a ML algorithm is a good practise for school seminars, but students' implementations are less efficient and usually introduce some bugs.

In [2]:
```
from sklearn import datasets
from sklearn import svm

iris = datasets.load_iris()
digits = datasets.load_digits()

clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(digits.data[:-1], digits.target[:-1])
```

Out[2]:
```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma=0.001, kernel='
rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=
True,
  tol=0.001, verbose=False)
```

ML models

- Supervised
  - Naive Bayes
  - Linear classifiers
  - Decision trees
  - Random forests
  - SVM
  - Neural networks
  - ...
- Unsupervised
  - Clustering algorithms (K-means, hierarchical clustering)
  - Neural networks
  - Outlier detection
  - ...

Supervised ML

- Typical for smaller data
- Data annotation is possible
- We know what we want to find/predict
- Not applicable usually for Facebook and Google like companies, but great for smaller companies and problems

Scenario 1

- I don't have time to implement/wait
- I don't need to explain results to somebody else
- It should perform reasonably well

  Try Naive Bayes

Naive Bayes

(sklearn source) Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector x_1 through x_n, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

In [3]:

```
import time
from sklearn import datasets
iris = datasets.load_iris()
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
start = time.time()
y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
print('Duration: %0.3f seconds' % (time.time() - start))
print("Number of mislabeled points out of a total %d points : %d"
       % (iris.data.shape[0],(iris.target != y_pred).sum()))
```

```
Duration: 0.001 seconds
Number of mislabeled points out of a total 150 points : 6
```

Scenario 2

- I have a lot of binary/multi value features (e.g. `color = {red, yellow, green}`)
- I want to explain decision
- Tuning should be intuitive

Try Decision trees

(sklearn source) Decision Trees are a supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

In [8]:

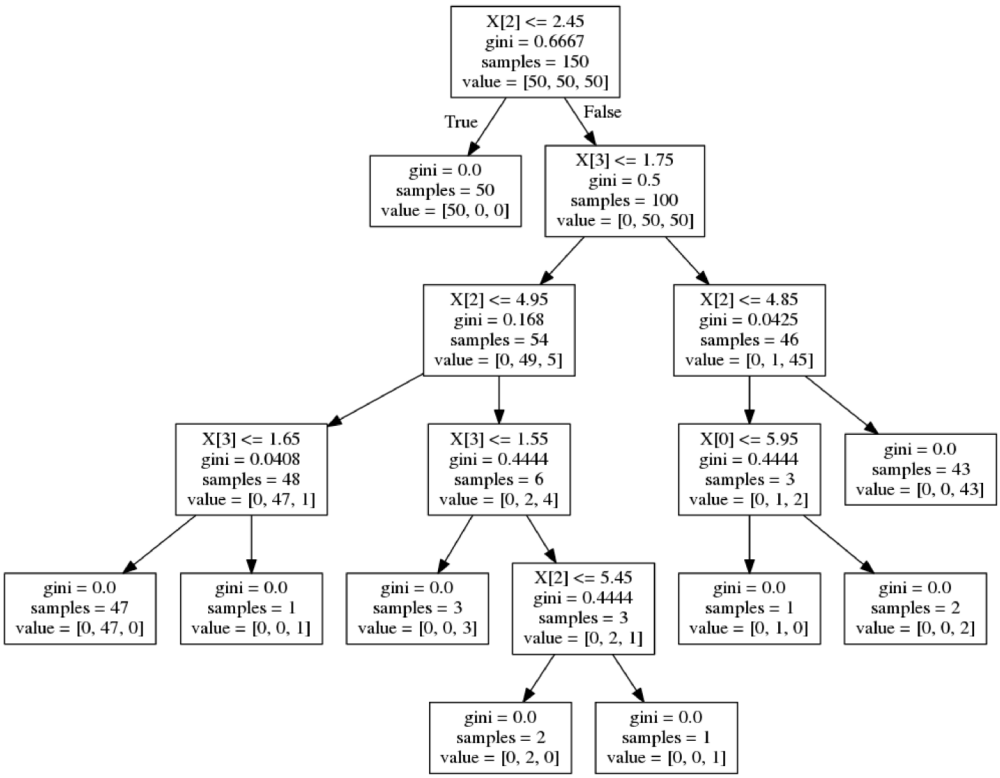
```
from sklearn.datasets import load_iris
from sklearn import tree

iris = load_iris()
clf = tree.DecisionTreeClassifier()
start = time.time()
clf = clf.fit(iris.data, iris.target)
print('Duration %0.3f seconds' % (time.time() - start))

import graphviz
dot_data = tree.export_graphviz(clf, out_file=None)
graph = graphviz.Source(dot_data, format='png')
graph.render("/tmp/i")

from IPython.display import Image
```

```
Duration 0.001 seconds
```

In [5]:

```
Image(filename='/tmp/i.png', width='880px')
```

Out[5]:



Scenario 3

If any of following aplies:

- I don't want **black box**, customer (court, security agency, clever CTO) wants to be explained decisions of system
- I can give time to output analysis
- I need quick performance

linear classifiers are perfect match.

In [22]:
```
from sklearn.linear_model import SGDClassifier
import time
from sklearn import datasets

iris = datasets.load_iris()

sgd = SGDClassifier()
start = time.time()
y_pred = sgd.fit(iris.data, iris.target).predict(iris.data)
print('Duration: %0.3f seconds' % (time.time() - start))
print("Number of mislabeled points out of a total %d points : %d"
      % (iris.data.shape[0], (iris.target != y_pred).sum()))
```

```
Duration: 0.002 seconds
Number of mislabeled points out of a total 150 points : 50
```

Analysis of trained weights:

- positive value: higher the value, higher the chance of label
- negative value: higher the absolute value, lower the chance of label
- select label with highest scalar product of weights and features

In [38]:
```
for no, label in enumerate(iris.target_names):
    print('Label %s' % label)
    print('; '.join([
        'weight[%s]=%0.2f' % (label, weight)
        for label, weight in zip(iris.feature_names, sgd.coef_[no]
)]))
```

```
Label setosa
weight[sepal length (cm)]=6.29; weight[sepal width (cm)]=29.16;
weight[petal length (cm)]=-48.60; weight[petal width (cm)]=-25.
16
Label versicolor
weight[sepal length (cm)]=48.03; weight[sepal width (cm)]=-153.
23; weight[petal length (cm)]=30.87; weight[petal width (cm)]=-
98.91
Label virginica
weight[sepal length (cm)]=-109.21; weight[sepal width (cm)]=-11
0.35; weight[petal length (cm)]=234.42; weight[petal width (cm)
]=167.52
```

Scenario 4

Bigger data set, previous methods don't work.

More powerfull techniques are needed:

- we want some readability: **random forests**
- black box is enough, but we don't have GPUs for NN: **SVM**

Random forests

- There are tools for explaining forests decisions (the most common paths in trees)
- Results can be analysed, human readable
- With increasing tree count and more deep structure, power of forest is growing
- Better for multi-value features
- Could horrible fail for periodic functions, e.g. **sinus**

In [47]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
import time
from sklearn import datasets

iris = datasets.load_iris()

clf = RandomForestClassifier(max_depth=2, random_state=0)
start = time.time()
clf.fit(iris.data, iris.target).predict(iris.data)
print('Feature weights:' % clf.feature_importances_)
print('Duration: %0.3f seconds' % (time.time() - start))
print("Number of mislabeled points out of a total %d points : %d"
       % (iris.data.shape[0], (iris.target != y_pred).sum()))
```

```
Feature weights:
Duration: 0.029 seconds
Number of mislabeled points out of a total 150 points : 37
```

Support vector machines

For long time, the best ML for NLP problems. Currently popularity of SVM is decreasing because of the deep learning.

Features of SVM:

- hard to select correct parameters (feature tuning is required)
- slower then previous methods
- except linear kernel works as black box
- more powerful, number of features can be higher than number of training samples

citing (sklearn):

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

Scenario 5

We have complex, difficult problem.

OR

We have images or other problems with binary features.

Use neural networks.

Neural networks

Steps:

- Think about features and how to extract them
  - E.g., using bag of words containing 1000 words, input layer consists of 1000 inputs
  - Each sentence is represented as a vector of zeros with several ones on positions corresponding to words in the sentence (one hot encoding)
  - Each sentence is represented as a vector of float numbers between 0 and 1, each word is encoded as a 1000 float numbers, each word encoding is different to each other word encodings
- Think how to design neural network:
  - how many layers
  - size of each layer
  - neural network type
  - which components should be used

Start with small and simple and add complexity.

Being expert in NN is a big project

- Testing hypothesis is very slow and you need good hardware
- Playing with many configurations, components, parameters, tricks, ...
- The best solutions are usually find by luck -- the space of all neural networks is too big
- Play with input data -- neural networks needs a lot of data -- generate data automatically, add noise to existing data, or use NN on unsupervised problems

Sklearn supports only the simplest neural networks, e.g.:

In [49]:
```
from sklearn.neural_network import MLPClassifier
```

Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

Neural networks

To use neural networks in Python, you have two reasonable possibilities:

- *Keras with Theano background:*
  - **Keras** is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.
  - It is more user friendly, but documentation is not great.
- *Tensorflow from Google:*
  - https://www.tensorflow.org/ (https://www.tensorflow.org/)
  - Ineffective on small number of GPUs
  - The most effective on big projects, many GPUs available
  - Harder to use and learn

Unsupervised algorithms

More creativity is needed. Manual result analysis is must.

Data/result analysis

Use some **baseline algorithm** to set up some expectations. E.g. guess labels randomly, guess the most common label.

If you are not happy with achieved results, analyse:

- your system (search bugs, use logging for each component, check manually and with asserts all components inputs and outputs)
- if system learns weights, check which features are used and which are ignored
- outputs: which classes are wrong, which are ok
- inputs: what happens with less inputs, with sampling, randomizing the order, ...
- features: do they cover all cases?

If results are better than you expected:

- again, check everything, chance of bug or using flawed dataset is big.

Everything is perfect:

- if you have time, still check.

Deploying trained model

Speeding up:

- more effective feature extraction
- better data structures
- using GPUs over CPUs for some techniques

Memory reduce:

- iterators instead of lists
- read structures from memory instead of holding it in RAM

Writing result parsers:

- converting ML outputs to desired format

Documentation and evaluation:

- achieved precision/recall/accuracy, strong and weak points of system
- recommended system configuration
- guess of experiment duration for sample data set lengths

Interface:

- good CLI or GUI, notebook, ...

In history, rewriting in C++, currently the speed is almost the same.

Questions?

Control checks

- You are analysing a mood of a person by first sentence in an opener message:

  ```
  Ahoj, máš se?
  Zapaříme?
  Neuvěříš, co se stalo...
  Blbej den! Já tu učitelku...
  čau vole:-)
  ```

  Recommend some features.
- You are given 100 annotated e-mails and 100 e-mails without labels to solve a classification problem. How do you divide data into training and testing set and tuning set?
- You have small project on classification of salesmans performance (influence of number of calls, appointments, overtimes and other factors on sales of examined person):
  - which ML technique will you use and why? Select one from NN, SVM, Random Forests, Decision Trees, Linear classifiers, Naive Bayes, ...

Thank you

In [51]:
```
# Start slideshow
!jupyter nbconvert notebook_demo.ipynb --to slides --post serve
```

```
[NbConvertApp] Converting notebook notebook_demo.ipynb to slide
s
[NbConvertApp] Writing 420106 bytes to notebook_demo.slides.htm
l
[NbConvertApp] Redirecting reveal.js requests to https://cdnjs.
cloudflare.com/ajax/libs/reveal.js/3.5.0
Serving your slides at http://127.0.0.1:8000/notebook_demo.slid
es.html
Use Control-C to stop this server
^C

Interrupted
```