pawel.kasprowski@polsl.pl

Artificial Intelligence

Data Mining, classification

Paweł Kasprowski, PhD, DSc.

Artificial Intelligence

- When there is no algorithmic solution of the problem
- The application ***learns*** to solve the problem
  - typically using some examples
- Intelligence is the ability to
  - learn based on given data
  - understand
  - use acquired knowledge to solve problems

Problems to be solved

- Classification
  - assign samples to predefined classes
  - [photo of an animal -> species]
  - [text -> author]
  - [scan-path -> novice/expert]
- Regression
  - calculate a value for each sample
  - [images of the house -> price of the house]
  - [eye image -> gaze coordinates]
- Conversion
  - convert object into another object
  - [image -> text description of the image]
  - [voice recording -> text of the speech]

Task and method

- Task: find a function Y = f(X) where
  - X – **sample** (input)
  - Y – result – class, value (output) – **label**
- Method (learning by example):
  - take some number of samples X with known output Y (**examples**, labeled samples)
  - build the function based on these examples (learning process)
  - use the function to predict the label for unknown samples

pawel.kasprowski@polsl.pl

Problem of "generality"

- It is relatively easy to create a function that correctly classifies all examples
- But is this function "general" enough?
  - is it able to correctly classify unknown samples?
- The answer is not trivial and that is why we have a lot of different classification methods
- "No free lunch" theorem

Over-fitting

- If the function (model) is optimized for the given data (given examples) it may have poor generality
- This problem is called over-fitting
- Solution is to simplify the model, e.g. stop learning even when it is not perfect for examples

The simplest algorithms

- K-Nearest Neighbors (kNN)
  - a new sample is classified as majority of its K nearest neighboring examples
- Linear Discriminant Analysis (LDA)
  - finds a linear equation separating positive and negative samples
- Naive Bayes
  - the class of a new sample is calculated based on examples using Bayes equation
- Decision Tree
  - the tree is built using examples by seaching for the most discriminative features
  - new samples are going through the tree reaching a leaf with the predicted class

More sophisticated algorithms

- Random Forest
  - a set of randomly generated decision trees voting for the final results
- Support Vector Machines (SVM)
  - recalculates examples into another space where they are easier to separate lineary (kernel trick)
- Neural Networks
  - based on neurons organized into layers

Hyperparameters

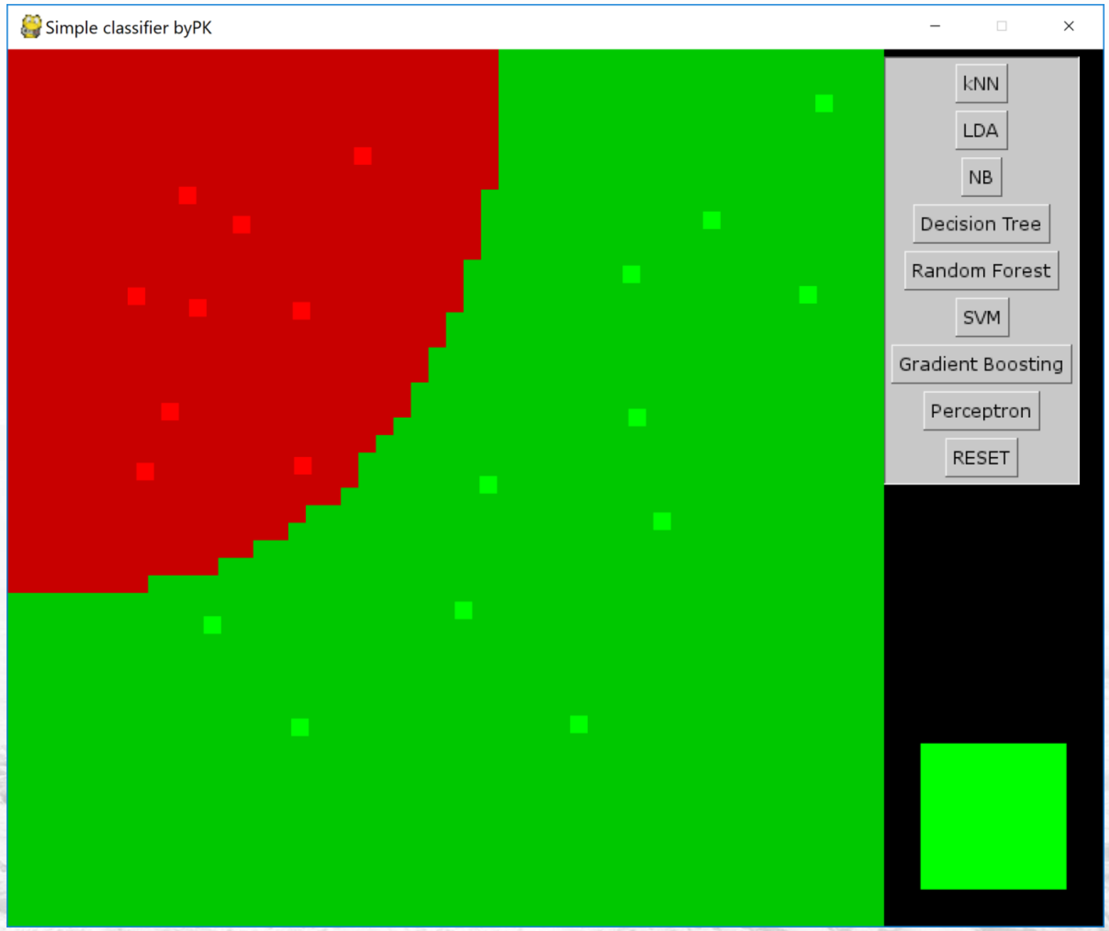
- Classification algorithms have parameters
  - (so called hyperparameters)
- kNN – number of neighbors
- Random Forest – number of trees
- SVM – kernel function, parameters of the function, learning rate etc.
- Good choice of hyper parameters influences the quality of the model!
- Bad news: deep learning methods have A LOT hyper parameters

Scikit-learn

- Package implementing:
  - many popular classification algorithms
  - many classic methods for data handling
- General use:
  - load the dataset consisting of samples and their labels
  - create a model using one of the possible algorithms (classes)
  - train the model: model.fit(samples, labels)
  - use the model to predict classes:
    - predicted = model.predict(sample)

Universal example

python classifier



Creating a model

```
def classification(model_name,samples,labels,rangex,rangey):
models = {
        "KNN": KNeighborsClassifier(),
        "LDA": LinearDiscriminantAnalysis(),
        "NB": GaussianNB(),
        "TREE":DecisionTreeClassifier(),
        "RF":RandomForestClassifier(n_estimators=20),
        "SVM":SVC(gamma='scale'),
        "PERC":Perceptron(max_iter=2000),
        "GB":GradientBoostingClassifier()
    }
model = models.get(model_name)
...
```

Using the model

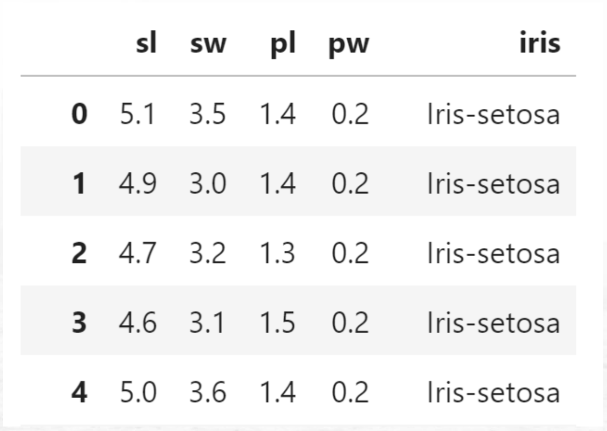
```
# training the model
samples = np.array(samples)
labels = np.array(labels)
model.fit(samples, labels)
# classify all points in the matrix
all_points = [(x,y) for x in range(rangex) for y in range(rangey)]
all_points = np.array(all_points)
r = model.predict(all_points)
# fill the matrix with predictions
result = np.zeros([rangex,rangey])
for i in range(len(all_points)):
        result[all_points[i,0],all_points[i,1]]=r[i]
return result
```

Example for the iris dataset

iris.ipynb

```
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
data = pd.read_csv('iris.data')

model = KNeighborsClassifier()
samples = data.values[:,0:4] # first four columns
labels = data.values[:,4] # the fifth column
model.fit(samples, labels)
predicted = model.predict([(5.9, 3.0, 5.1, 1.8)])
print("predicted",predicted)
> predicted ['Iris-virginica']
```



	sl	sw	pl	pw	iris
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Evaluating results

- Hyperparameters may be tuned
  - it is important to be able to check if the model is general
- First rule: never check the model using the same data that you used for training (examples)
  - over-fitting!
- The dataset should always be divided into:
  - training set (used to build the model)
  - test set (to check how it works for unknown samples)

Evaluation

- Dividing into training set and test set may be not enough!
  - we optimize the model (by tuning hyper parameters) for the given test set
- The safest way:
  - training set (for learning)
  - validation set (for hyper parameters tuning)
  - test set (for final evaluation)
    - the "holdout" dataset

Sklearn train_test_split

- (trainSamples, testSamples, trainLabels, testLabels) = sklearn.model_selection.train_test_split(samples, labels)

iris.ipynb / Train Test Split

- Two pairs:
  - trainSamples, trainLabels
  - testSamples, testLabels
- Default: 75% train, 25% test
- Parameters:
  - train_size – percent or number of samples
  - test_size – percent or number of samples

Stratification

- Problem: proportion of samples for classes may differ in training and test sets
- For instance: 50-50-50 in the original set
  - training: 40-30-25
  - test: 10-20-25
- Effect: class0 will be treated as more probable!
- Solution: stratified division (maintain distribution)
- Parameter: stratify=labels

Check test samples

iris.ipynb / Check an error

- Checking the test set

```
correct = 0;
predictedLabels = model.predict(testSamples)
for i in range(len(testSamples)):
    print(testLabels[i],"->",predictedLabels[i],end=' ')
    if(testLabels[i]==predictedLabels[i]):
        correct = correct + 1; print('OK')
    else:
        print('error!!!')
print("Correct: {} of {} accuracy = {:.2f}"
      .format(correct,len(testSamples),correct/len(testSamples)))
```

Cross-validation

- Dynamic division into training and test
  - the same examples are sometimes training and sometimes test samples
- 10-fold cross validation:
  - divide the whole dataset into 10 subsets
  - for each subset
    - train the model using the remaining 9 subsets
    - test the results using the chosen subset
  - average the results
- Folds are randomized but may be stratified
  - the same distribution of classes in each fold

Sklearn cross_validate

- ***iris.ipynb / Cross validation***
  - sklearn.model_selection.cross_validate(model, samples, labels, cv=<number of folds>)
- Returns for each fold:
  - fit_time
  - score_time
  - test_score
- Many parameters to check other metrics!

Measures

- Accuracy – the number of correctly classified samples to all samples
  - a problem with unbalanced sets
  - if 90% of test samples is positive the blind classifier achieves 90% accuracy
- Precision – the cost of false positives
  - not belonging to class predicted as belonging
- Recall – the cost of false negatives
  - belonging to class predicted as not belonging
- F1-Score – combination of precision and recall
- Cohen's kappa – compares prediction with random prediction

Measures interpretation

- Example: We have built the model that diagnoses COVID-19 based on a genetic test
- For every eye movement sample the model returns:
  - ***H*** - if a person is healthy
  - ***S*** - when a person is sick
- We test the model using
  - 18 samples from sick people (***S***)
  - 82 samples from healthy people (***H***)
- We achieve 82% accuracy
  - is it bad or good?
  - it depends: we must examine a confusion matrix

Confusion matrix

Predicted as >> Real class	H	S
H	81	1
S	17	1

P	N
TP	FN
FP	TN

Accuracy 0.82

H:
precision: 0.83
recall: 0.99
F1-score: 0.90

S:
precision: 0.50
recall: 0.06
F1-score: 0.1

Cohen's kappa: 0.07

Measures

Predicted as >> Real class	H	S
H	80	2
S	15	3

P	N
TP	FN
FP	TN

Accuracy 0.83

H:
precision: 0.84
recall: 0.98
F1-score: 0.90

S:
precision: 0.60
recall: 0.17
F1-score: 0.26

Cohen's kappa: 0.20

Measures

Predicted as >> Real class	H	S
H	66	16
S	2	16

P	N
TP	FN
FP	TN

Accuracy 0.82

H:
precision: 0.97
recall: 0.80
F1-score: 0.88

S:
precision: 0.50
recall: 0.89
F1-score: 0.64

Cohen's kappa: 0.53

Measures

Predicted as >> Real class	H	S
H	64	18
S	0	18

P	N
TP	FN
FP	TN

Accuracy 0.82

H:
precision: 1.00
recall: 0.78
F1-score: 0.88

S:
precision: 0.50
recall: 1.00
F1-score: 0.67

Cohen's kappa: 0.56

Example for three classes

Confusion matrix

```
[[24  3  9]
 [ 3 24 24]
 [ 2 15 49]]
```

class	precision	recall	f1-score	support
0	0.83	0.67	0.74	36
1	0.57	0.47	0.52	51
2	0.60	0.74	0.66	66
micro avg	0.63	0.63	0.63	153
macro avg	0.67	0.63	0.64	153
weighted avg	0.64	0.63	0.63	153

Accuracy: 0.63

Scikit learn measures

- Useful functions from *sklearn.metrics*:
  - confusion_matrix(labels, results)
  - classification_report(labels, results)
  - accuracy_score(labels, results)
  - precision_score(labels, results)
  - recall_score(labels, results)
  - cohen_kappa_score(labels, results)
  - ...

Measures calculation

iris.ipynb / Measures calculation

```
from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score,cohen_kappa_score

predictedLabels = model.predict(testSamples)
print(confusion_matrix(testLabels, predictedLabels))
print(classification_report(testLabels, predictedLabels))
accuracy = accuracy_score(testLabels, predictedLabels)
print("Accuracy: {:.2f}".format(accuracy))
c_kappa = cohen_kappa_score(testLabels, predictedLabels)
print("Cohen's Kappa: {:.2f}".format(c_kappa))
```

Adding own scorers to cross validation

iris.ipynb / Adding own scorers to cross validation

```
from sklearn.metrics import make_scorer, precision_score, recall_score

p_scorer = make_scorer(precision_score, average='micro')
r_scorer = make_scorer(recall_score, average='micro')
my_scorer = {'precision': p_scorer, 'recall': r_scorer}

sklearn.model_selection.cross_validate(model, samples, labels, cv=5,
    scoring=my_scorer)
```

Summary

- Accuracy is often not enough
  - especially for unbalanced datasets
- Cohen's kappa is the reliable way to evaluate results
  - but has some drawbacks as well!
- The evaluation depends on the target we want to achieve
  - False Acceptance Rate (FAR=0)
    - no incorrectly classified healthy people
  - False Rejection Rate (FRR=0)
    - no incorrectly classified sick people