BCH2203 Python - 10. Machine Learning

Ramses van Zon

20 March 2024

This class

- We'll do a lightning overview of some of the **machine learning** you can do in Python.
- Machine Learning is a bit like an adventurous form of statistics.
- Will briefly look at:
  - regression
  - classification
  - cluster analysis

  (so not deep learning - sorry)
- Take away message: use the `scikit-learn` package

About Scikit-learn

- Standard machine learning package in Python.
- Get it with
  `from sklearn import ...`
  or
  `from sklearn.SUBPACKAGE import ...`
- Built on numpy, scipy, and matplotlib.
- Can do regression, classification, clustering, decision trees, ...
- https://scikit-learn.org

Regression

Linear Regression

- Imagine you have data in the form of samples of variables $x_1, x_2, x_3, \ldots$ and one (or more) variable(s) y.

x_1	x_2	x_3	$\ldots$	y
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

- You fit this data to a model $y = f(x_1, x_2, x_3, \ldots)$, where f has fitting parameters.
- For linear regression, f is linear in the parameters and one allows for randomness.
- After regression, you can use the model to make predictions about y given new data $x_1, x_2, \ldots$.

Statistics and Machine learning call things differently here: . . .

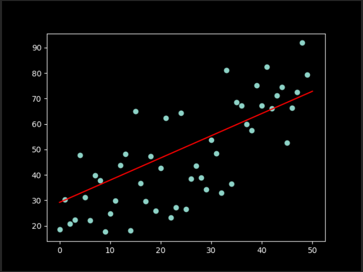
- In machine learning, you call the x values **features**, and y values **labels**.
- In statistics, you can the x values the **independent variables** and y values **dependent variables**

Linear Regression in Python

- Independent variable: x
- Dependent variable: y
- Assume $y = ax + b$ plus noise
- a is the coefficient
- b is the offset.

First, let's generate data:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> n = 50
>>> x = np.arange(float(n))
>>> y = x + 50*np.random.random(n)
>>> plt.plot(x,y)
```



Data points $(x_i, y_i) \rightarrow$ best estimate a and b.

- Possible with just numpy:

```
>>> fit = np.polyfit(x, y, 1)
>>> print(fit)
[  0.87283243  29.21160988]
>>> predict = np.polyval(fit,12.5)
>>> print(predict)
40.122015255
```

```
>>> plt.plot([0,50],
             [np.polyval(fit,0),
              np.polyval(fit,50)],
             'r')
```

- Or with scikit-learn

Scikit-learn version

```
>>> from sklearn import linear_model

>>> def column(a):
...     """turn a vector into a column matrix."""
...     return np.expand_dims(a,-1)

>>> regr = linear_model.LinearRegression()

>>> regr.fit(column(x), column(y))
>>> print(regr.coef_, regr.intercept_)
[[ 0.87283243]] [ 29.21160988]

>>> print(regr.predict(column([12.5])))
[[ 35.8567218]]
```

This is typical in Machine Learning:

- Train a model, i.e., fit its parameters on data.
- Make predictions given new data.

- All input and output are 2D: this way, one can fit multiple *features* (x) and *targets* (y).

  I.e., even for a single vector, `sklearn` wants a matrix.

- `column(a)` is a helper function to turn a vector into column matrix.

  If a is a numpy array, then `b=column(a)` and `b=a[:,None]` are equivalent.
  And the reverse would be `a=b[:,0]`.

- Use `fit()` to train the model,
  and `predict()` to apply it to (new) data.

- Polynomial fits require creating x^2, x^3 as additional independent variables (sklearn.preprocessing.PolynomialFeatures)

We still don't quite know how well we did.

- For this, we need another typical machine-learning approach:
  Dividing the data in a 'training' and a 'test' set.
- In general, we get our data, and that's it. We don't have the luxury of just generating more data.
- We would like to do out-of-sample testing of whatever model we generate, to see how it does against new data. But we don't have any new data.
- The solution is to hold out some of the original data. Most of the data is used for training the model, the rest is used for testing it. These data should be chosen randomly, as in the next slide.

Separating training and testing data

We generated data already:

```
>>> import numpy as np
>>> n = 50
>>> x = np.arange(float(n))
>>> y = x + 50*np.random.random(n)
```

Now we'll set a side some of that data for testing:

```
>>> test_fraction = 0.2
>>> test_selection = np.random.random(n) < test_fraction
>>> x_test, y_test = x[test_selection,None], y[test_selection,None]
```

And train/fit using the rest:

```
>>> train_selection = np.logical_not(test_selection)
>>> x_train, y_train = x[train_selection,None], y[train_selection,None]
>>> regr = linear_model.LinearRegression()
>>> regr.fit(x_train, y_train)
```

Goodness of fit

- We have fitted on the **training data**.
- We can now see how well this works for the test data.
- For instance, we could use the (built-in) R^2 metric on the predictions for the **test data**:

```
>>> print(regr.score(x_test,y_test))
0.567106344383
```

The closer the R^2 score is to 1, the better the fit.

- More metrics in `sklearn.metrics`

Train/Test Split using scikit-learn

Generating data (as before):

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> n = 50
>>> x = arange(float(n))
>>> y = x + 50*np.random.random(n)
```

Splitting using sklearn:

```
>>> from sklearn.model_selection import train_test_split
>>> test_fraction = 0.2
>>> x_train, x_test, y_train, y_test = train_test_split(x[:,None],y[:,None],test_size=test_fraction)
```

Training as before:

```
>>> regr = linear_model.LinearRegression()
>>> regr.fit(x_train, y_train)
```

Scoring on the test portion of the data:

```
>>> print(regr.score(x_test, y_test))
0.596961546282
```

Nothing like a visual confirmation.

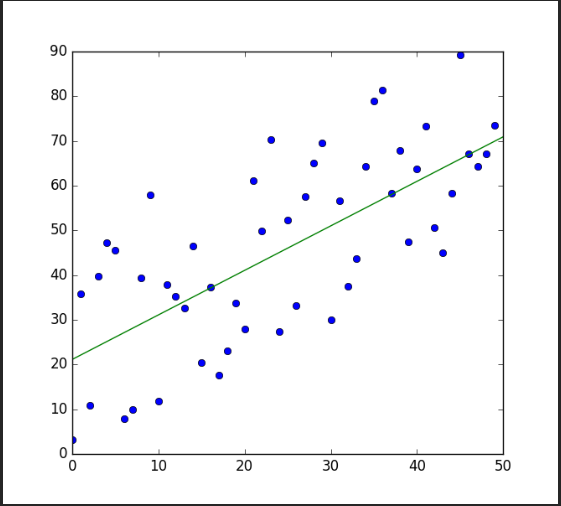
R^2=0.596961546282 looks neither very good nor too bad.

Nothing like a plot to see the result:

```
>>> import matplotlib.pylab as plt
>>> from numpy import linspace
>>> px = linspace(0.0,float(n),200)
>>> py2d = regr.predict(px[:,None])
>>> py = py2d[:,0]
>>> plt.plot(x,y,'o',px,py,'-')
>>> plt.show();plt.pause(.1)
```

Remember: matplotlib needs 1d arrays, and sklearn uses 2d arrays.

- 1d to 2d: `px` $\rightarrow$ `px[:,None]`.
- 2d to 1d: `py2d` $\rightarrow$ `py2d[:,0]`.



Classification

Classification Basics

Classification is similar to regression, in a sense:

- You fit a model to data with known answers ($y = f(x_1, x_2, x_3, \ldots)$).
- You use the model to make predictions about new data.

But what do you do if the labels (y) are discrete? How do you deal with that?

- Data point y is either in category 1 or 2.
- You don't get points for putting y in category 1.5.

Classification algorithms are used to create models for separating data into known categories.

Classification in Python

Some classic classification problems:

- Bioinformatics - classifying proteins according to function.
- Medical diagnosis
- Image processing:
  - what objects exist in an image?
  - hand-written text analysis.
- Text categorization:
  - Spam filtering
  - Sentiment analysis: is this tweet positive or negative?
- Language recognition.
- Fraud detection.

Input variables can be continuous, discrete, or both.

Classification approaches

There are lots of classification approaches which one might use.

- *Decision trees:* analyze the features of the data and make 'decisions' about how to 'split' the data into uniform groups.
- *Logistic regression:* like linear regression, but now we fit a "yes/no" function to the data.
- *Naive Bayes:* a type of probabilistic analysis.
- *kNN:* k Nearest Neighbours; use the k nearest neighbours to a data point to predict the category of a new data point.
- *Support Vector Machines:* essentially a linear model of the data, used for separate groups.
- *Neural networks:* an algorithmic approach to using functions to categorize data.

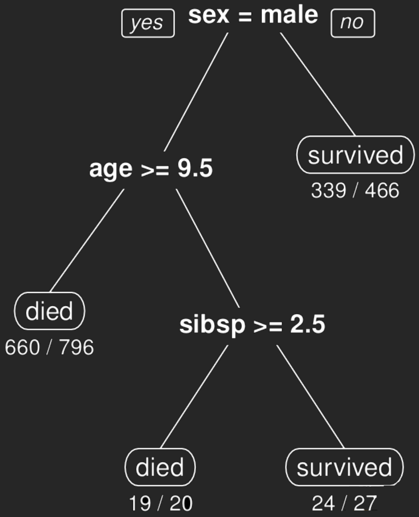
There isn't time to cover all of these. Let's look at **Decision Trees**.

Decision Trees

A Decision Tree is a structure which classifies an input based on a number of binary decisions.

It splits the data set based on one of the p "features" of the data.

"Features" are the independent variables associated with the data $(x_1, x_2, \ldots, x_p)$.



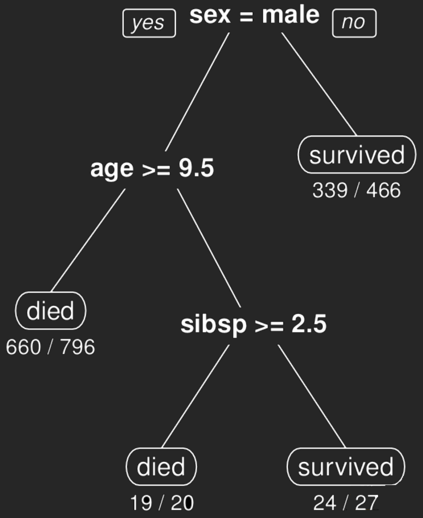
Decision Trees, continued

Data can be split based on discrete data

("if category == A") or continuous data

("if height < 1.5m")

The goal of developing a decision tree is to determine when and where and how to split the data, so as to maximize the 'purity' of the resulting sub-data set.



Splitting algorithms

Algorithms which split the data, wil rank possible splits based on increasing ‘purity’ of the two subgroups it generates, and try to find the best one.

Splitting algorithms

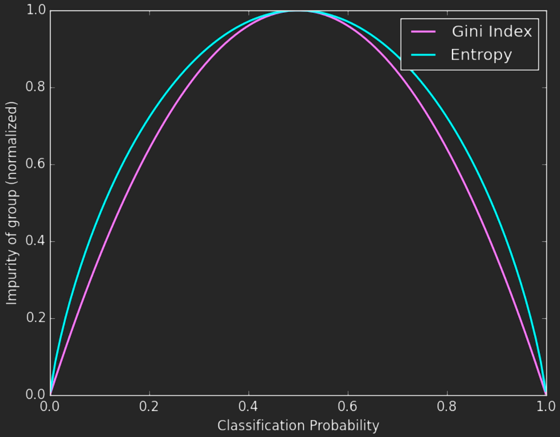
Algorithms which split the data, wil rank possible splits based on increasing 'purity' of the two subgroups it generates, and try to find the best one.

Consider the probability p that a member of one of the labels is in a given feature category. Two common measures for the 'impurity' of the generated groups are given by

- Gini index: $\sum p(1-p)$
- Entropy: $-\sum[p\ln p + (1-p)\ln(1-p)]$

where the sum is over all labels and possible values in the given category.

An impurity of 0, i.e., probability of 0 or 1, is perfect.



Splitting algorithms, continued

Splitting algorithms proceed iteratively.

While every data point is not in a pure sub-tree:

- For each feature in the data remaining in the sub-tree, consider a split:
    - If the feature is categorical, consider all values, split by value and measure the impurity of the resulting subgroups.
    - If the feature is continuous, use line optimization to choose the best point at which to split, keeping track of the impurity at that point.
- Choose the split which maximizes the change in the impurity (smallest impurity value), and split the data.

Example: Iris data

Let's use sklearn to build a decision tree. We'll use the Iris data set.

- The data consists as four measurements of 150 wild irises of 3 species.
- It's a classic classification problem.
- It's one of the data sets which comes with sklearn.
- We first randomly split the iris data set, 70/30, into training and test data sets.

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> iris = datasets.load_iris()
>>> train_data, test_data, train_target, test_target = train_test_split(
        iris.data, iris.target, test_size=0.3)
```

https://en.wikipedia.org/wiki/Iris_flower_data_set

Example: Iris data, continued

Now that the data's split up, we're ready to generate the tree.

- import the `DecisionTreeClassifier`
- Specify which features to use
- Generate the tree.
- Check against the training data.
- Pretty good fit!
- How about test data?

```
>>> from sklearn.tree import DecisionTreeClassifier
>>>
>>> iris_tree = DecisionTreeClassifier(
...      criterion = "gini", random_state = 1,
...      max_depth=4, min_samples_leaf=5)
>>>
>>> iris_tree.fit(train_data, train_target)
>>>
>>> print(iris_tree.score(train_data, train_target))
0.971428571429
```

```
>>> print(iris_tree.score(test_data, test_target))
0.9555555555555556
```

Not bad!

Confusion matrix

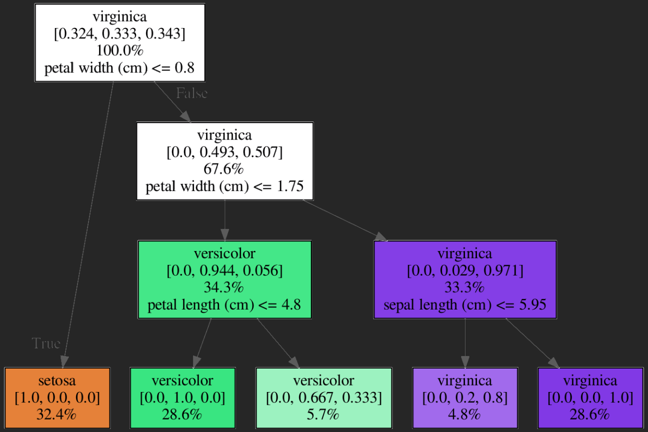
To ascertain the effectiveness of a classifier when labels are known, we can look at the confusion matrix.

```
>>> from sklearn.metrics import confusion_matrix
>>>
>>> ypred = iris_tree.predict(test_data)
>>> ytrue = test_target
>>>
>>> m = confusion_matrix        (ypred,ytrue)
>>>
>>> print(m)
[[11  0  0]
 [ 0 16  1]
 [ 0  1 16]]
```

Mostly diagonal; only one mislabeling: very good!

Plot a decision tree

```
>>> from sklearn.tree import plot_tree
>>> plot_tree(iris_tree,
  feature_names=iris.feature_names,
  class_names=iris.target_names,
  proportion=True,
  impurity=False,
  filled=True);
```



Trees and over-fitting

As with polynomials and regression, we can easily produce overly-complex decision trees which do great on the training data, but don't generalize.

In fact, this is guaranteed to happen with decision trees, since given enough splits, it will always perfectly classify the data.

How do we deal with this? The usual approach is to prune the tree at some level, where the results are "good enough", and the model is not "too complex".

Cluster Analysis

Clustering

Clustering is classification without the classes.

- Unsupervised learning - no labels.
- Assign groups of “similar” observations to the same cluster.

Scientific applications:

- Assign proteins with similar interactions to same group
- Find patterns in galaxy properties
- Determine topics in bodies of text

Business applications

- Market segmentation
- “People who buy X often buy. . . ”

Clustering

Two primary reasons for clustering:

- Uncover undiscovered patterns in high-dimensional data
- Summarize large number of observations into fewer, homogeneous clusters.

Definition of “similar”, “cluster” notably vague.

Typically involve short “distances” between points in the p-dimensional space of features.

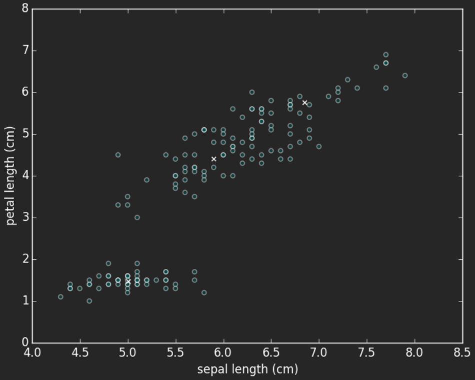
Continuous spaces - a Euclidean or other distance metric.

Ordinal spaces (e.g., bag-of-word counts): use a ‘cosine similarity’.

K-means

K-means clustering is a geometric clustering algorithm which uncovers roughly spherical blobs of clusters amongst the data items. The algorithm is very simple:

- Starting with k initial cluster centers,
- For each data point, assign to nearest centre,
- Calculate the centroid of each new cluster,
- Move cluster centers to new centre,
- Repeat until converged

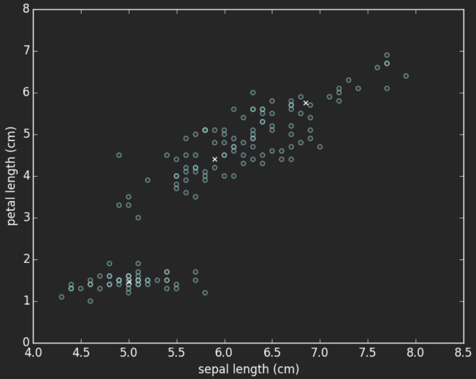


K-means: pros and cons

K-means is extremely robust, but has some downsides:

- Have to know before hand how many (k) clusters you're looking for.
- Random initial positions can go badly wrong;
- Need many initial tries; handled automatically by `k-means`

How to measure quality of clusters?



K-means: Error measures

A few error measures available for k-means:

- Homogenity: how similar are in-cluster items?

  This involves something like minimizing the within-cluster sum of squares

$$\mathrm{WCSS} = \sum_i^k \sum_{j \in S_k} ||x - \mu_j||^2$$

- Completeness: how different are items in one cluster from items in another?

  This involves something like maximizing the between-cluster sum of squares

$$\mathrm{ICSS} = \sum_i^n \sum_j^n \delta\left(S_i, S_j\right) ||x_i - x_j||^2$$

K-means with scikit-learn

- Import KMeans from sklearn.cluster.
- Instantiate a cluster algorithm with 3 cluster.
- Use fit() to fit data.

  Note: we are not using the labels (i.e., target).
- Use predict to predict the categories of new data.

```
>>> from sklearn import datasets
>>> from sklearn.cluster import KMeans
>>> iris = datasets.load_iris()
>>>
>>> kmeans = KMeans(n_clusters=3, random_state=0)
>>> kmeans.fit(iris.data)
>>>
>>> print(kmeans.predict([[6.2,2.7,6.5,1.9]]))
[2]
```

What would be a good way to ascertain the correctness?

That's right, we should split train/test data and see how well the test data get predicted by the model obtained from the train data.