Introduction to Machine Learning

Alexey Fedoseev

March 12, 2024

Machine learning: supervised and unsupervised

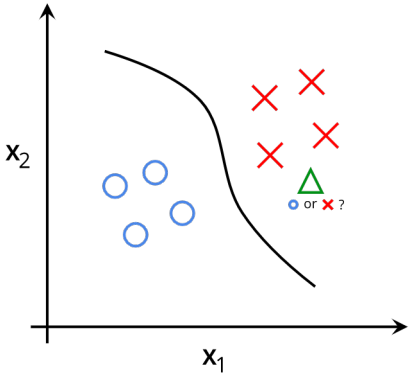
What is machine learning?

Machine learning algorithms build a mathematical model of sample data in order to make predictions or decisions without being explicitly programmed to perform the task.

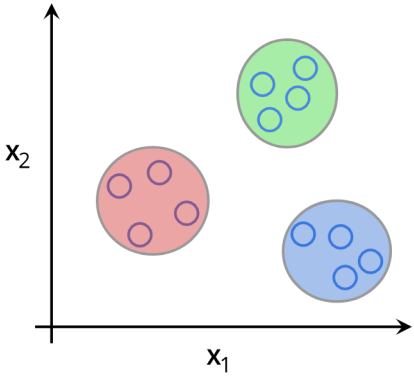
When we're working with data, we generally have two types of analyses:

- Supervised: the data comes labeled with the right answer:
  - curve fitting.
  - for prediction-type analyses (decision trees, neural networks,...)
- Unsupervised: we're looking for patterns in the data:
  - what groups of items in this dataset are similar? Dissimilar?
  - Generally used for exploration, evaluation and sometimes prediction.
- There is also semi-supervised, but we won't be dealing with that.

Supervised Learning



Unsupervised Learning



Types of Data

Generally speaking, data comes in two broad classes:

- Continuous: real numbers
- Discrete:
  - Binary: True/False.
  - Categorical: category A, category B, ...
  - Ordinal: discrete, but has an intrinsic order: S, M, L, XL, ...

Others are possible too, but we won't be covering them.

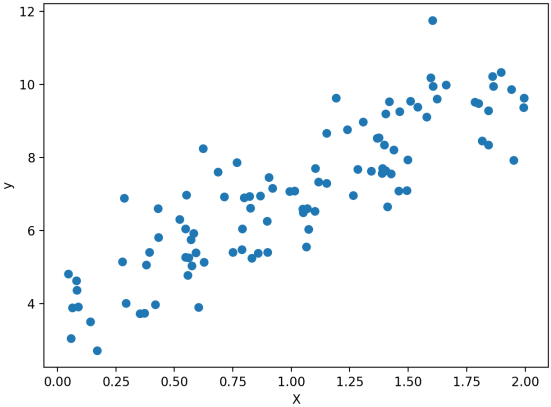
Regression

Regression is a central part of machine learning. In machine learning we are not so concerned with how well the regression model fits to the data, but rather care about how well it predicts new observations.

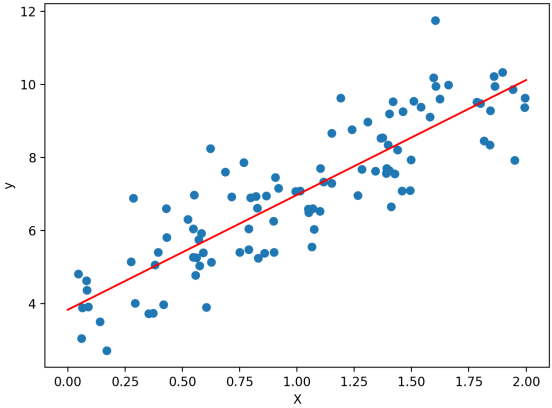
- Data comes as a set of n observations, each of which has p `features`.
- We will assume continuous features (not always the case).
- The goal is to learn the function $y = \hat{f}(x_1, x_2, ..., x_p)$ for predicting new values.

Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(36)

X = 2*np.random.rand(100,1)
y = 4+3*X + np.random.randn(100,1)*0.9

plt.scatter(X, y)
plt.xlabel("X")
plt.ylabel("y")
```



```python
import sklearn.linear_model as slm
lin_reg = slm.LinearRegression()
lin_reg.fit(X, y)

xx = np.linspace(0,2,100).reshape(-1,1)
yy = lin_reg.predict(xx)

plt.scatter(X, y)
plt.plot(xx, yy, "r")
```



Polynomial Regression

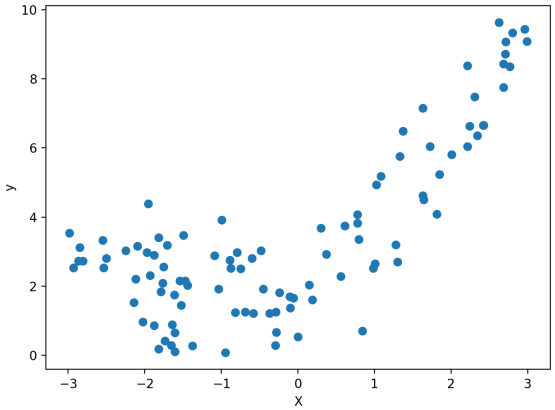
What if your data is actually more complex than a simple straight line?

You can actually use a linear model to fit nonlinear data.

- Add powers of each feature as new features
- Train a linear model on this extended set of features.

This technique is called Polynomial Regression.

```python
import numpy as np
import matplotlib.pyplot as plt

m = 100

X=6*np.random.rand(m,1)-3
y=0.5*X**2+X+2+np.random.randn(m,1)*0.9

plt.scatter(X, y)
plt.xlabel("X")
plt.ylabel("y")
```

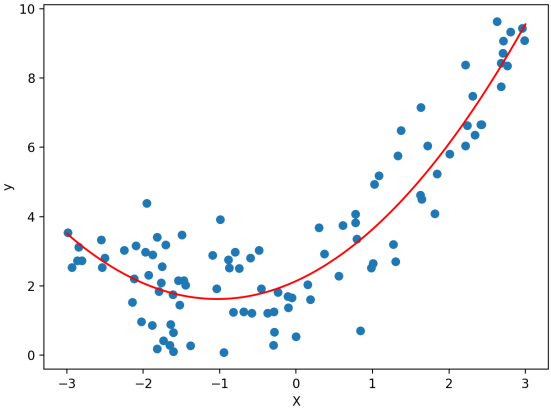


```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
poly_feat = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_feat.fit_transform(X)
print(poly_feat.get_feature_names())
print(X_poly[0], X[0])
```

Output

```
['x0', 'x0^2']
[1.65717363 2.74622444] [1.65717363]
```

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)

print(lin_reg.coef_,lin_reg.intercept_)

xx=np.linspace(-3,3,100).reshape(-1,1)
yy=lin_reg.predict(
    poly_feat.transform(xx))

plt.scatter(X, y)
plt.plot(xx, yy, "r")
```



Output

```
[[1.05702383 0.53440749]] [1.87683274]
```

Originaly we had

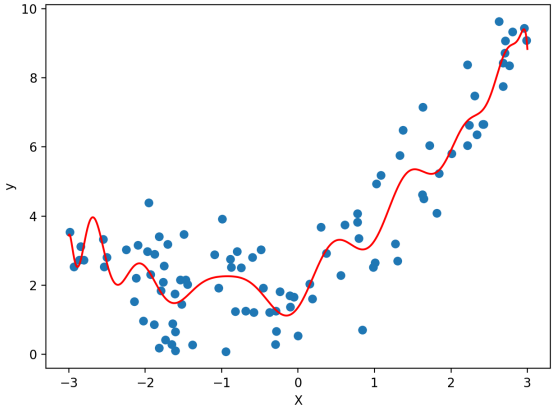
```
y = 0.5 * X**2 + X + 2 + np.random.randn(m,1) * 0.9
```

The model predicted

```
y_pred = 0.53440749 * X**2 + 1.05702383 * X + 1.87683274
```

Overfitting

```
poly_feat = PolynomialFeatures(
    degree=20, include_bias=False)
X_poly = poly_feat.fit_transform(X)

lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)

xx=np.linspace(-3,3,1000).reshape(-1,1)
yy=lin_reg.predict(
    poly_feat.transform(xx))

plt.scatter(X, y)
plt.plot(xx, yy, "r")
```



Transformation Pipelines

As we saw with the Polynomial Regression there are could be many data transformation steps that need to be executed in the right order. Scikit-Learn provides the Pipeline class to help with such sequences of transformations

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
    ("lin_reg", LinearRegression()),
])

polynomial_regression.fit(X, y)
```

Classification

Like regression, classification is a central topic in machine learning. It deals with the discrete target labels. Examples of these labels are:

- Yes or No
- Male or Female
- Salary brackets: below \$40k, \$40k-\$60k, \$60k-\$80k, \$80k-\$100k, \$100k+

Decision Tree

To understand Decision Trees, let's build one and take a look at how it makes predictions. Let's use the famous 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.datasets
- The data comes as an sklearn `bunch`
- We first randomly split the data set, 66.6%/33.3%, into training and test data sets

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

Decision Tree

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

iris = load_iris()
print(iris.DESCR)
```

```
...
:Number of Instances: 150
    (50 in each of three classes)
:Number of Attributes: 4 numeric,
    predictive attributes and the class
:Attribute Information:
    - sepal length in cm
    - sepal width in cm
    - petal length in cm
    - petal width in cm
    - class:
        - Iris-Setosa
        - Iris-Versicolour
        - Iris-Virginica
...
```

Training versus Testing

In general, we get our data, and that's it. We don't have the luxury of generating more data on a whim.

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.

Decision Tree

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.33, random_state=42)
```

Now that the data is split up, we are ready to generate the tree.

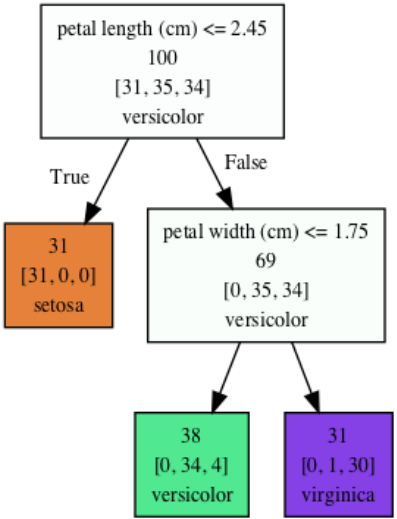
```
from sklearn import tree, metrics

model = tree.DecisionTreeClassifier(max_depth=2)
model.fit(X_train, y_train)
```

Decision Tree

It's always good to plot your decision tree.
Install `graphviz` package

```
$ conda install python-graphviz
```

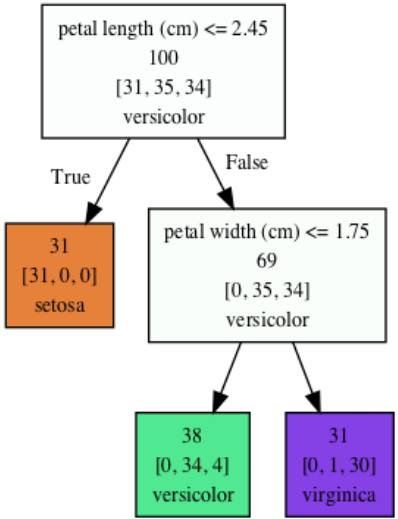
```
import graphviz
dot_data = tree.export_graphviz(
    model, out_file=None,
    class_names=iris.target_names,
    feature_names=iris.feature_names,
    impurity = False, filled = True,
    label = 'none')
graph = graphviz.Source(dot_data)
graph.format = 'pdf'
graph.render("iris", view=True)
```



Decision Tree

You found an iris flower and you want to classify it. Start at the root node (depth 0).

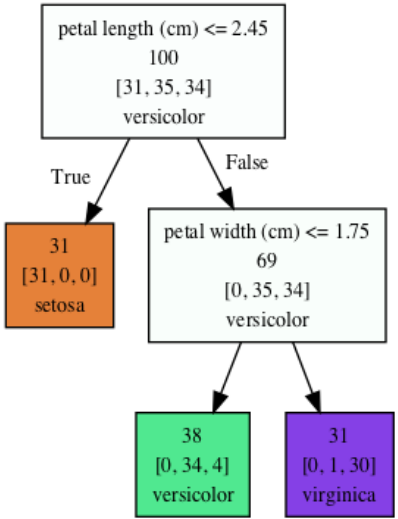
- Measure the petal length of flower
- If it is smaller than 2.45cm you move down to the left node (depth 1)
- The left node of the Decision Tree predicts that your flower is Iris-Setosa.



Decision Tree

Let's say you have another flower you want to classify. You start at the root node (depth 0).

- Measure the petal length of flower
- If it is greater than 2.45cm you move down to the right node (depth 1)
- Is the petal width smaller than 1.75 cm?
- If it is, then your flower is most likely an Iris-Versicolor (depth 2, left)
- If not, it is likely an Iris-Virginica (depth 2, right).



Confusion Matrix

How to determine the effectiveness of a classifier?

A good way to evaluate the performance of a classifier is to look at the confusion matrix. The general idea is to count the number of times instances of class A are classified as class B.

```
y_pred = model.predict(X_train)
print(metrics.confusion_matrix(y_train, y_pred))
```

```
array([[31,  0,  0],
       [ 0, 34,  1],
       [ 0,  4, 30]])
```

	setosa (predicted)	versicolor (predicted)	virginica (predicted)
setosa (actual)	31	0	0
versicolor (actual)	0	34	1
virginica (actual)	0	4	30

Decision Tree

How does the decision tree do on the test data?

```
y_pred = model.predict(X_test)
print(metrics.confusion_matrix(y_test, y_pred))
```

```
array([[19,  0,  0],
       [ 0, 15,  0],
       [ 0,  1, 15]])
```

Only once virginica was misclassified as versicolor.

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".