Applied Machine Learning
Lecture 4-1: Linear models

Selpi (selpi@chalmers.se)

The slides are further development of Richard Johansson's slides

January 31, 2020

Overview

Linear classifiers and the perceptron

Linear regressors

Linear classifier

- Has scoring function looks like:

$$\text{score} = \boldsymbol{w} \cdot \boldsymbol{x}$$

- where
  - $\boldsymbol{x}$ is a vector with features of what we want to classify (e.g. made with a `DictVectorizer`)
  - $\boldsymbol{w}$ is a vector representing which features the classifier thinks are important
  - $\cdot$ is the dot product between the two vectors
- there are two classes: **binary** classification
  - return the first class if the score > 0
  - ... otherwise the second class
- the essential idea: **features are scored independently**

quick note

- sometimes, linear classifiers are expressed as

$$\text{score} = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

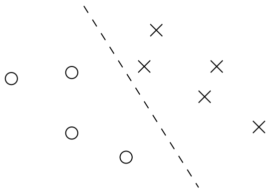
where b is an "offset" or intercept

interpreting linear classifiers

- each weight in $\boldsymbol{w}$ corresponds to a feature
  - e.g. `"fine"` probably has a high positive weight in sentiment analysis
  - `"boring"` a negative weight
  - `"and"` near zero

Geometric view

- Linear classifiers are a class of geometric models.
- Geometrically, a linear classifier is a classifier that use hyperplanes to separate/classify the examples/vector space into two regions
  - using lines to separate 2-dimensional data
  - using planes to separate 3-dimensional data



example: plotting the decision boundary in scikit-learn

- See notebook

training linear classifiers

- the family of learning algorithms that create linear classifiers is quite large
  - perceptron, Naive Bayes, support vector machine, logistic regression/MaxEnt, ...
- their underlying theoretical motivations can differ a lot but in the end they all return a weight vector $\boldsymbol{w}$

the perceptron learning algorithm

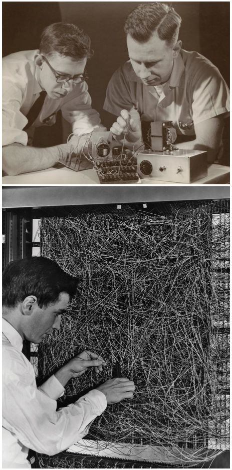
- start with an empty weight vector (all zeros)
- repeat: classify according to the current weight vector
- each time we misclassify, change the weights a bit
  - if a positive instance was misclassified, add its features to the weight vector
  - if a negative instance was misclassified, subtract ...

the perceptron learning algorithm, formally

$\boldsymbol{w} = (0, \ldots, 0)$ $\quad$ ($\leftarrow$ a weight vector of all zeros)
repeat N times
  for $(\boldsymbol{x}_i, y_i)$ in the training set
    score $= \boldsymbol{w} \cdot \boldsymbol{x}_i$
    if a positive instance is misclassified
      $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{x}_i$
    else if a negative instance is misclassified
      $\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{x}_i$
return $\boldsymbol{w}$

a historical note

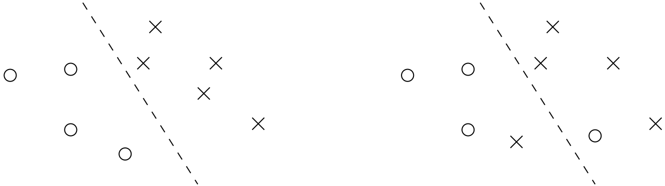
- the perceptron was invented in 1957 by Frank Rosenblatt
- initially, a lot of hype!
- the realization of its limitations led to a backlash against machine learning in general
  - the nail in the coffin was the publication in 1969 of the book *Perceptrons* by Minsky and Papert
- new hype in the 1980s, and now...



[source]

linear separability

- a dataset is **linearly separable** if there exists a $\boldsymbol{w}$ that gives us perfect classification

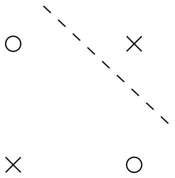


- **theorem**: if the dataset is linearly separable, then the perceptron learning algorithm will find a separating $\boldsymbol{w}$ in a finite number of steps
- what if the dataset is not linearly separable?

a simple example of linear inseparability

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

very good	Positive
very bad	Negative
not good	Negative
not bad	Positive



coding a linear classifier using NumPy

```
class LinearClassifier(object):

    def predict(self, x):
        score = x.dot(self.w)
        if score >= 0.0:
            return self.positive_class
        else:
            return self.negative_class
```

better: handle all instances at the same time

```
class LinearClassifier(object):

    def predict(self, X):
        scores = X.dot(self.w)
        out = numpy.select([scores>=0.0, scores<0.0],
                           [self.positive_class,
                            self.negative_class])
        return out
```

side note: what happened in that code?

```
>>> import numpy

>>> scores = numpy.array([-1, 2, 3, -4, 5])

>>> scores >= 0
array([False,  True,  True, False,  True], dtype=bool)

>>> scores < 0
array([ True, False, False,  True, False], dtype=bool)

>>> numpy.select([scores >= 0, scores < 0], ["positive", "negative"])
array(['negative', 'positive', 'positive', 'negative', 'positive'],
      dtype='|S8')
```

https://docs.scipy.org/doc/numpy-1.15.4/reference/generated/numpy.select.html

perceptron reimplementation in NumPy

```
class Perceptron(LinearClassifier):
    def __init__(self, n_iter=10):
        self.n_iter = n_iter

    def fit(self, X, Y):

        n_features = X.shape[1]
        self.w = numpy.zeros( n_features )

        for i in range(self.n_iter):
            for x, y in zip(X, Y):

                score = self.w.dot(x)

                if score <= 0 and y == self.positive_class:
                    self.w += x
                elif score >= 0 and y == self.negative_class:
                    self.w -= x
```

still too slow...

- this implementation uses NumPy's dense vectors
- with a large training set with lots of features, it may be better to use SciPy's sparse vectors
- however, $\boldsymbol{w}$ is a dense vector and I found it a bit tricky to mix sparse and dense vectors
- this is the best solution I've been able to come up with for the two operations $\boldsymbol{w} \cdot \boldsymbol{x}$ and $\boldsymbol{w} {+}{=} \boldsymbol{x}$

```
def sparse_dense_dot(x, w):
    return numpy.dot(w[x.indices], x.data)

def add_sparse_to_dense(x, w, xw):
    w[x.indices] += xw*x.data
```

reimplementation with sparse vectors

```
class SparsePerceptron(LinearClassifier):

    # ...

    def fit(self, X, Y):
        # ... some initialization

        for i in range(self.n_iter):
            for x, y in zip(X, Y):

                score = sparse_dense_dot(x, self.w)

                if score <= 0 and y == self.positive_class:
                    add_sparse_to_dense(x, self.w, 1)
                elif score >= 0 and y == self.negative_class:
                    add_sparse_to_dense(x, self.w, -1)
```

Linear classifiers in scikit-learn

- small selection of learning algorithms:
  - `sklearn.linear_model.Perceptron`
  - `sklearn.linear_model.LogisticRegression` (lecture 5)
  - `sklearn.svm.LinearSVC` (lecture 5)
- to compute the score function: `model.decision_function(x)`
- after training, weights are stored in the attribute `model.coef_`

Overview

Linear classifiers and the perceptron

Linear regressors

linear regression models

- a **linear regression** model predicts numerical output values like this:

$$y = \boldsymbol{w} \cdot \boldsymbol{x}$$

- explanation of the parts:
  - again, $\boldsymbol{x}$ is an encoded feature vector
  - $\boldsymbol{w}$ is a vector representing relationships between features and the output y

recall: analytic solution to least squares regression

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

recall: analytic solution to least squares regression

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}$$

- not the way it's implemented in scikit-learn: any idea why?

the Widrow-Hoff algorithm

- almost like the perceptron!
- NB the "learning rate" η

$\boldsymbol{w} = (0, \ldots, 0)$
repeat N times
  for $(\boldsymbol{x}_i, y_i)$ in the training set
    $g = \boldsymbol{w} \cdot \boldsymbol{x}_i$
    error $= g - y_i$
    $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \text{error} \cdot \boldsymbol{x}_i$
return $\boldsymbol{w}$

Linear regressors in scikit-learn

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model