Machine Learning for NLP
Lecture 3: Linear classifiers

UNIVERSITY OF GOTHENBURG

Richard Johansson

September 10, 2015

this lecture

- classifiers in vector spaces: **linear classifiers**
  - perceptron
  - support vector machines
  - logistic regression
  - ...
- implementation in NumPy/SciPy
- introduction to the second assignment

overview

linear classifiers

case study: the perceptron

training linear classifiers with optimization

introduction to assignment 2

recap: basic vector operations

the basic operations on vectors:

- **scaling**: $\alpha \cdot \mathbf{v} = \alpha \cdot (v_1, \ldots, v_n) = (\alpha \cdot v_1, \ldots, \alpha \cdot v_n)$
- **addition** and **subtraction**:
  $\mathbf{v} + \mathbf{w} = (v_1, \ldots, v_n) + (w_1, \ldots, w_n) = (v_1 + w_1, \ldots, v_n + w_n)$
- **scalar product** or **dot product**:
  $\mathbf{v} \cdot \mathbf{w} = (v_1, \ldots, v_n) \cdot (w_1, \ldots, w_n) = v_1 \cdot w_1 + \ldots + v_n \cdot w_n$
- **vector length** or **norm**:
  $|\mathbf{v}| = |(v_1, \ldots, v_n)| = \sqrt{v_1 \cdot v_1 + \ldots + v_n \cdot v_n} = \sqrt{\mathbf{v} \cdot \mathbf{v}}$

linear classifiers

- a **linear classifier** is a classifier that is defined in terms of a scoring function like this

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

- explanation of the parts:
  - $\boldsymbol{x}$ 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
- for now, we'll assume that 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**

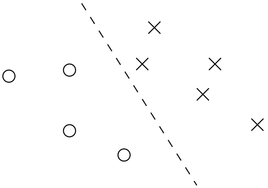
UNIVERSITY OF GOTHENBURG

example

“a really good movie”

geometric view

- geometrically, a linear classifier can be interpreted as separating the vector space into two regions with a line (plane, hyperplane)



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 $\boldsymbol{w}$

a linear classifier in NumPy/scikit-learn

```
class LinearClassifier(BaseEstimator, ClassifierMixin):
    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

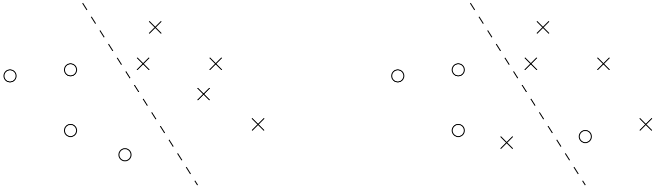
```
class LinearClassifier(BaseEstimator, ClassifierMixin):
    def predict(self, X):
        scores = X.dot(self.w)
        out = numpy.select([scores>=0.0, scores<0.0], [self.positive_class,
                                                        self.negative_class])
        return out
```

an illustration of the steps

```
>>> 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')
```

linear separability

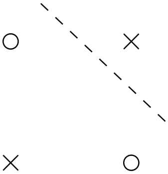
- a dataset is **linearly separable** if there exists a $\boldsymbol{w}$ that gives us perfect classification



- theorem: if the dataset is linearly separable, then the perceptron learning algorithm will find a separating $\boldsymbol{w}$ in a finite number of steps

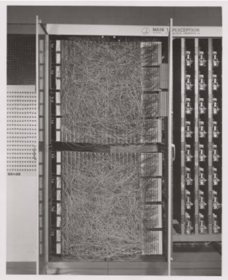
a simple example of linear inseparability

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



a historical note

- the perceptron was invented in 1957 by Frank Rosenblatt
  - here's an image (from Wikipedia) of the first implementation
- 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...

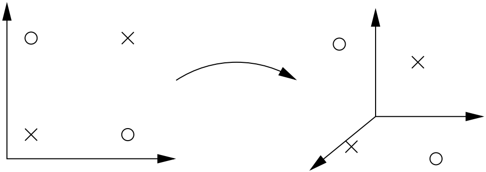


mapping into a larger vector space

- we may add "useful combinations" of features to make the dataset separable:

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

- from a geometrical viewpoint: we are creating a feature space with a higher dimensionality:



- lots of features $\rightarrow$ LOTS of combinations

overview

linear classifiers

case study: the perceptron

training linear classifiers with optimization

introduction to assignment 2

recap: our simple perceptron implementation

- start with an empty weight table
- go through examples, classify according to the current weights
- each time we misclassify, change the weight table a bit
  - if a positive instance was misclassified, add 1 to the weight of each feature in the document
  - and conversely . . .

recap: our simple perceptron implementation

- start with an empty weight table
- go through examples, classify according to the current weights
- each time we misclassify, change the weight table a bit
  - if a positive instance was misclassified, add 1 to the weight of each feature in the document
  - and conversely . . .

```
def perceptron_learn(examples, number_iterations):
    weights = {}
    for iteration in range(number_iterations):
        for label, features in examples:
            guess = perceptron_classify(features, weights)
            if label == "pos" and guess == "neg":
                for f in features:
                    weights[f] = weights.get(f, 0) + 1
            elif label == "neg" and guess == "pos":
                for f in features:
                    weights[f] = weights.get(f, 0) - 1
    return weights
```

vector formulation of the perceptron algorithm

- start with an empty weight vector: $\mathbf{w} = (0, 0, \ldots, 0)$
- go through examples, classify according to the current weights
  - $score = \mathbf{w} \cdot \mathbf{x}$
- each time we misclassify, change the weight vector a bit
  - if a positive instance was misclassified, add 1 to the weight of each feature in the document: $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  - and conversely . . . : $\mathbf{w} = \mathbf{w} - \mathbf{x}$

reimplementation in NumPy/scikit-learn

```
class Perceptron(LinearClassifier):

    def __init__(self, n_iter=10):
        self.n_iter = n_iter

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

        X = X.toarray() # convert sparse to dense
        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
```

a reformulation of the perceptron algorithm

- in many machine learning papers, the positive and negative class are implicitly represented as +1 and -1, respectively
- then the perceptron algorithm can be written a bit more compactly

```
class Perceptron(LinearClassifier):
    # ...
    def fit(self, X, Y):
        # ... some initialization

        for i in range(self.n_iter):

            for x, y in zip(X, Y):

                score = self.w.dot(x) * y

                if score <= 0:
                    self.w += y*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 $\boldsymbol{w} \mathrel{+}= \boldsymbol{x}$

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

def add_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) * y

                if score <= 0:
                    add_sparse_to_dense(x, self.w, y)
```

comparison

- on my computer, with the data set we'll use in assignment 2:
  - dense vectors: 17 seconds
  - sparse vectors: 3 seconds

overview

linear classifiers

case study: the perceptron

training linear classifiers with optimization

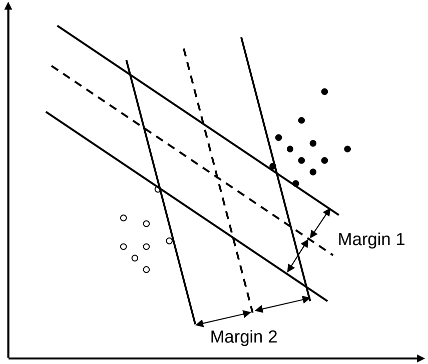
introduction to assignment 2

optimization and machine learning

- we will now consider models that are less ad-hoc than the perceptron
- idea: define an **objective function** based on the fundamental tradeoff in machine learning:
  - how well we handle the training set (**loss**)
  - simplicity of the model (**regularization**)
- ... and then the training consists of applying optimization techniques such as gradient descent to find the best $\boldsymbol{w}$
- we will consider two models:
  - support vector classifier, based on geometry
  - logistic regression, based on probability

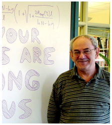
margin of separation

- the **margin** γ denotes how well $\boldsymbol{w}$ separates the classes:



large margins are good

- a result from statistical learning theory:

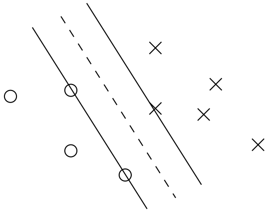


$$\text{expected test error} = \text{training error} + \text{BigUglyFormula}\left(\frac{1}{\gamma^2}\right)$$

- larger margin $\rightarrow$ better generalization

support vector machines

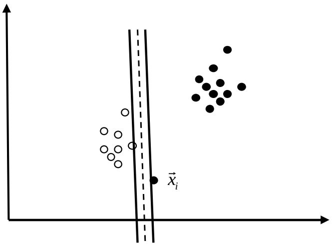
- **support vector machines** (SVMs) or support vector classifiers (SVC) are linear classifiers constructed by selecting the $\boldsymbol{w}$ that maximizes the margin



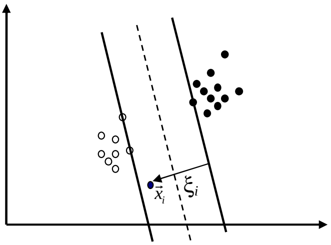
- note: the solution depends only on the borderline examples: the **support vectors**
- note: this solution is unique, while e.g. perceptron depends on initialization and processing order

soft-margin SVMs

- in some cases the dataset is inseparable, or nearly inseparable
- **soft-margin SVM**: allow some examples to be disregarded when maximizing the margin



A) Hard Margin SVM



B) Soft Margin SVM

implementing the SVM

- the hard-margin and soft-margin SVM can be stated mathematically in a number of ways
- also, the mathematical formulation leads to an optimization problem, which can be addressed in many different ways
  - general-purpose optimization software
  - specialized algorithms (usually better)
- more details later

linear classifiers with probabilities?

- we'll consider a linear classifier that is based on probabilities rather than geometry
- linear classifiers select the outputs based on a scoring function:

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

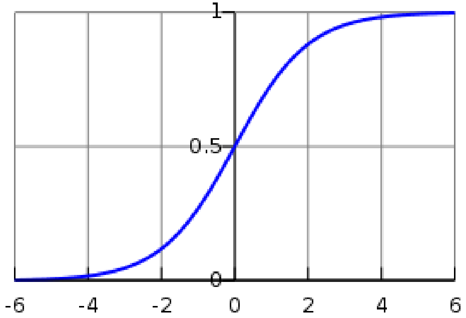
- how to convert the scores into probabilities?
- idea: use a **logistic function**:

$$P(\text{positive output}|\boldsymbol{x}) = \frac{1}{1 + e^{-\text{score}}}$$

where e^{-score} = `math.exp(-score)`

- this is formally a probability: always between 0 and 1, sum of probablities of possible outcomes = 1

the logistic function



logistic regression

- we find the best $\mathbf{w}$ by **maximum likelihood**: select the parameters to make our dataset maximally probable
- we can train a linear classifier by adjusting $\mathbf{w}$ to maximize the probability of our training set:

$$L(\mathbf{w}) = P(y_1|\mathbf{x}_1) \cdot \ldots \cdot P(y_T|\mathbf{x}_T)$$

- this model is called **logistic regression**
- it is equivalent to the **maximum entropy** classifier

in scikit-learn

- SVM is called `sklearn.svm.LinearSVC`
- LR is called `sklearn.linear_model.LogisticRegression`

stating SVM and LR formally

- SVM and LR come from different mathematical backgrounds
- however, using a few mathematical tricks, it can be shown that they both can be written in this form

$$\min_{\boldsymbol{w}} \lambda |\boldsymbol{w}|^2 + \frac{1}{m} \sum_{\boldsymbol{x},y} \text{Loss}(\boldsymbol{w}, \boldsymbol{x}, y)$$

- the **loss** function checks how well the classifier fits the training set:
  - for SVM: $\max(0, 1 - y \cdot \text{score})$ ("**hinge loss**")
  - for LR: $\log(1 + \exp(-y \cdot \text{score}))$ ("**log loss**")
- the first part is a **regularizer** that keeps the classifier simple
  - λ controls the tradeoff between the loss and the regularizer
  - some formulations use C instead of λ, with the opposite effect

overview

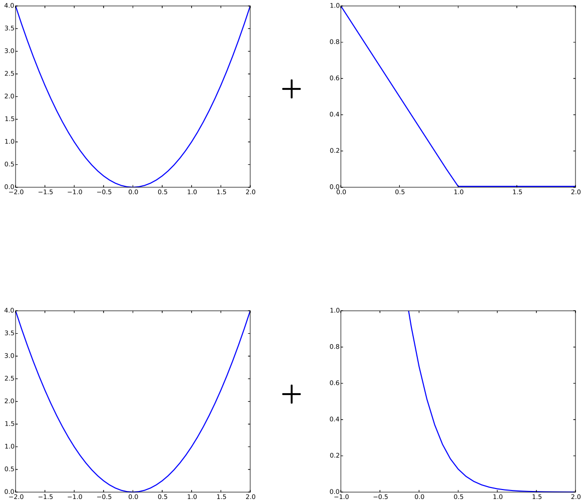
linear classifiers

case study: the perceptron

training linear classifiers with optimization

introduction to assignment 2

SVM and LR have convex objective functions



recap: stochastic gradient descent

- since the objective functions of SVM and LR are convex, we can find $\boldsymbol{w}$ by gradient descent
- **stochastic gradient descent**: like gradient descent, but we just compute the gradient for a single example
- pseudocode:
  1. set $\boldsymbol{w}$ to some initial value, e.g. all zero
  2. if we are "done", terminate and return $\boldsymbol{w}$
  3. otherwise, select a single training instance $\boldsymbol{x}$
  4. select a "suitable" step length η
  5. compute the gradient $\nabla f(\boldsymbol{w})$ **using $\boldsymbol{x}$ only**
  6. subtract $\eta \cdot \nabla f(\boldsymbol{w})$ from $\boldsymbol{w}$ and go back to step 2

some comments about assignment 2

- implement the SVM and test it in a document classifier
- we'll use the **Pegasos** algorithm – see assignment page
- Pegasos works in an iterative fashion similar to the perceptron
  - . . . so if you start from my perceptron code this will be a breeze
- for VG, three additional requirements:
  - a small trick to speed up one part of the implementation
  - your code should work with sparse feature vectors
  - you should you implement logistic regression as well

some clarifications about the paper

- the important part of the paper is the pseudocode in Figure 1
- Pegasos adapts the step length η over time: long steps in the beginning, smaller in the end
- $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$ is the dot product $\boldsymbol{w} \cdot \boldsymbol{x}$
- S is the training set, $|S|$ is the size of S
- T is the number of steps in the algorithm.
  - this is a bit different from our perceptron, where we specified the number of times to process the whole training set.
- the optional line is there for theoretical reasons and can be ignored
- a **subgradient** is a gradient for “abrupt” functions such as the hinge loss

practical information

- solve the assignment individually
- two lab sessions next week
- deadline one week later: **September 24**

seminar next week

- we need two “voluntary” students or pairs to present research papers at the seminar next week (September 18)
- possible topics:
  - predicting a suitable learner level for a sentence (Ildikó’s work)
  - selecting the correct preposition (“I believe **in** Santa Claus”)