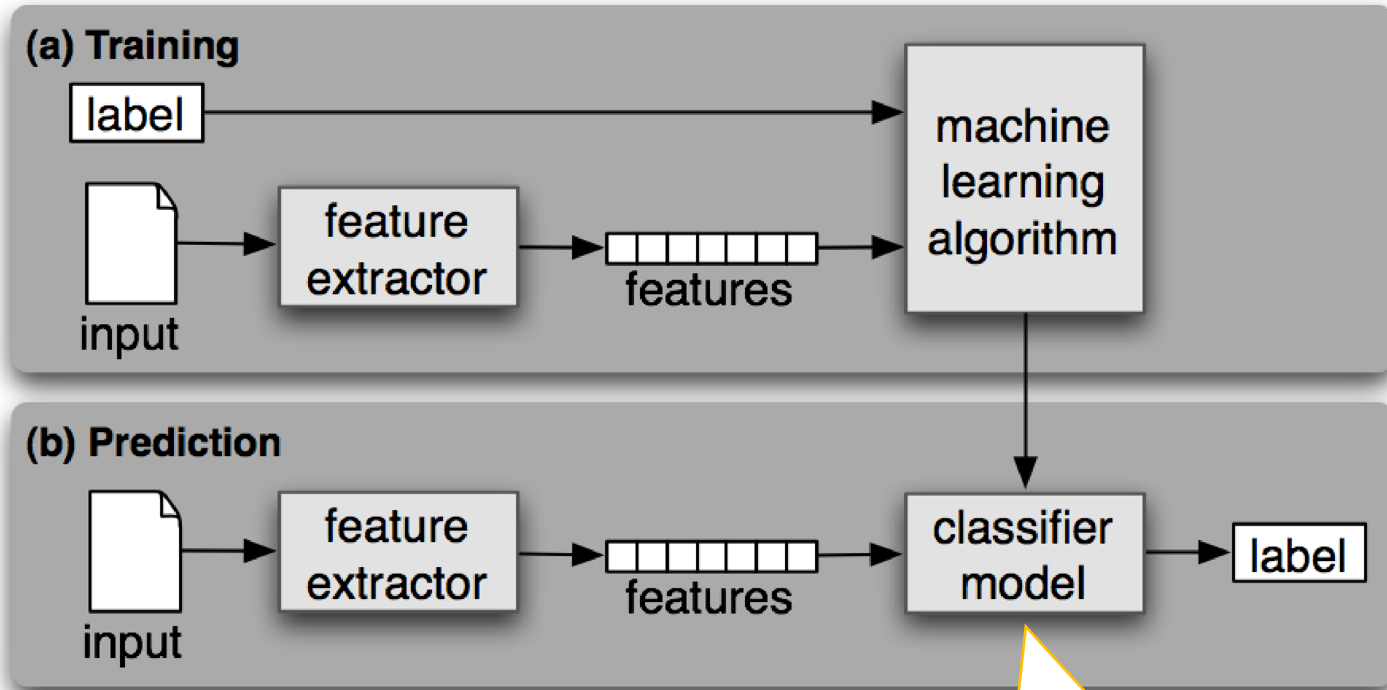
Text classification

IN4080
Natural Language Processing

Yves Scherrer

Supervised classification



Today:
- **Perceptron**
- **Logistic regression**

Supervised classification

General prediction function:

$$\hat{y} = \arg\max_{y \in Y} S(x, y)$$

- $x \in X$: input instance
- $y \in Y$: class/label
- $\hat{y}$: predicted class according to a model
- S is some scoring function and depends on the type of classification algorithm

- Prediction function for probabilistic models:

$$\hat{y} = \arg\max_{y \in Y} P(y|x)$$

Bayesian inference

- Prediction: $\hat{y} = \arg\max_{y \in Y} P(y|x)$
- Bayes' theorem: $P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$
- Thus: $\hat{y} = \arg\max_{y \in Y} \frac{P(x|y) \cdot P(y)}{P(x)}$
- $P(x)$ does not affect the argmax computation
- Thus: $\hat{y} = \arg\max_{y \in Y} (P(x|y) \cdot P(y))$

Properties of Naïve Bayes

- A probabilistic classifier
  - There are also non-probabilistic ones
- A multi-class classifier
  - Can handle more than two classes
  - This is the default case for NLP problems
- Uses batch training:
  - Each training instance is seen exactly once
  - The order in which training instances are seen does not matter
  - Probabilities can be computed exactly (“closed form”), there is no random/non-deterministic element in the computation

Perceptron

Perceptron

Why not forget about probabilities and learn the weights in an error-driven way?

$$\hat{y} = \arg\max_{y \in Y} S(x, y)$$

Training:

- Take one instance x of the training set
- Predict a label $\hat{y}$ using the current model
- If the prediction is correct, nothing happens
- If the prediction is wrong, modify the parameters of the model
- Continue “until tired” (J. Eisenstein)

Perceptron

- The perceptron algorithm starts with a default model, which is then continuously adjusted and improved.
- There is no natural end point of the training process.
  - You may want to see every training instance at least once (one **epoch**), but you're not required to.
  - There are heuristics to figure out when it is a good moment to stop.
- Results will vary depending on the initial model and the order of presenting the instances.

Bag-of-words representations for the perceptron

- Change of notation:
  - x_k: the kth instance of the dataset
  - f_i: the ith word in the vocabulary
    - This was x_i last week…
  - $f_{i,k}$: the frequency of word i in instance k

- Count features:
  - $f_{1,k} \equiv$ the number of times the word *Ronaldo* occurs in x_k
- Binary features:
  - $f_{1,k} \equiv 1$ if x_k contains the word *Ronaldo*, 0 otherwise

Count features and binary features may even be combined in the same model. Statistical independence is not required.

The perceptron prediction function

The perceptron associates each feature f_i with a weight w_i:

- The feature values change with each instance
- The weight values change with each class

The perceptron scoring function:

$$S(x_k, y) = \sum_{i}^{n} w_{i,y} \cdot f_{i,k}$$

- where n is the number of features (vocabulary size)

The perceptron prediction function

The weights and features can be rewritten as vectors:

- $\boldsymbol{w_y} = [w_{1,y}, w_{2,y}, \dots, w_{n,y}]$
- $\boldsymbol{f_k} = [f_{1,k}, f_{2,k}, \dots, f_{n,k}]$

and the scoring function can be rewritten as their **dot product**:

$$S(x_k, y) = \boldsymbol{w_y} \cdot \boldsymbol{f_k}$$

Putting everything together, the perceptron prediction function is thus:

$$\hat{y} = \arg\max_{y \in Y} (\boldsymbol{w_y} \cdot \boldsymbol{f_k})$$

The perceptron prediction function

What about smoothing?

- Possible, but usually not required.
- There is no danger of canceling out the entire dot product, as its main operation is addition.

Many types of classifiers use the same prediction function ($\boldsymbol{w} \cdot \boldsymbol{f}$)

- The particularity of the perceptron lies in the approach to estimate the values of the weight vector.

Perceptron training

Prediction function:

$$\hat{y} = \arg\max_{y \in Y}(\boldsymbol{w_y} \cdot \boldsymbol{f_k})$$

How to learn $\boldsymbol{w}$ with the perceptron algorithm:

- Take one instance x_k of the training set
- Predict a label $\hat{y}$ using the current model
- If the prediction is correct, nothing happens
- If the prediction is wrong, modify $\boldsymbol{w}$ How exactly?
- Continue "until tired" (J. Eisenstein) When exactly?

At the beginning, $\boldsymbol{w}$ is typically initialized to 0.

Update

If the prediction is correct ($\hat{y} = y$), nothing happens:

- $\boldsymbol{w}_y$ does not change.
- The weight vectors of the other classes do not change either.

If the prediction is wrong ($\hat{y} \neq y$):

- The weight vector of the correct label y is updated by adding the feature values:
$$\boldsymbol{w}_y \leftarrow \boldsymbol{w}_y + \boldsymbol{f}_k$$
- The weight vector of the predicted (but wrong) label $\hat{y}$ is updated by subtracting the feature values:
$$\boldsymbol{w}_{\hat{y}} \leftarrow \boldsymbol{w}_{\hat{y}} - \boldsymbol{f}_k$$
- The weight vectors of the other classes do not change.

Example

- Three classes: A, B, C
- Four features: f_1, f_2, f_3, f_4
- Weight vectors w_A, w_B, w_C initialized to 0
- In case of ties, predict the alphabetically first class

- First training instance: $[4, 0, 0, 2]$, class C
- Second training instance: $[0, 1, 0, 1]$, class B
- Third training instance: $[1, 0, 7, 0]$, class A
- Fourth training instance: $[0, 2, 0, 0]$, class B

Algorithm: Training and testing

```
procedure train_perceptron (D):
    w_y = [0…0] for all labels y          # weight vectors initialized to zeros
    repeat:
        for each document x with label y in D:
            f_x = extract_feature_vector(x)      ⎫
            y_hat = argmax_y' ( w_y' · f_x )     ⎭ test_perceptron (x, w)
            if y_hat != y:
                w_y = w_y + f_x
                w_y_hat = w_y_hat – f_x
            end if
        end for
    until stopping condition met
return w
```

```
procedure test_perceptron (x, w):
    f_x = extract_feature_vector(x)
    y_hat = argmax_y' ( w_y' · f_x )
return y_hat
```

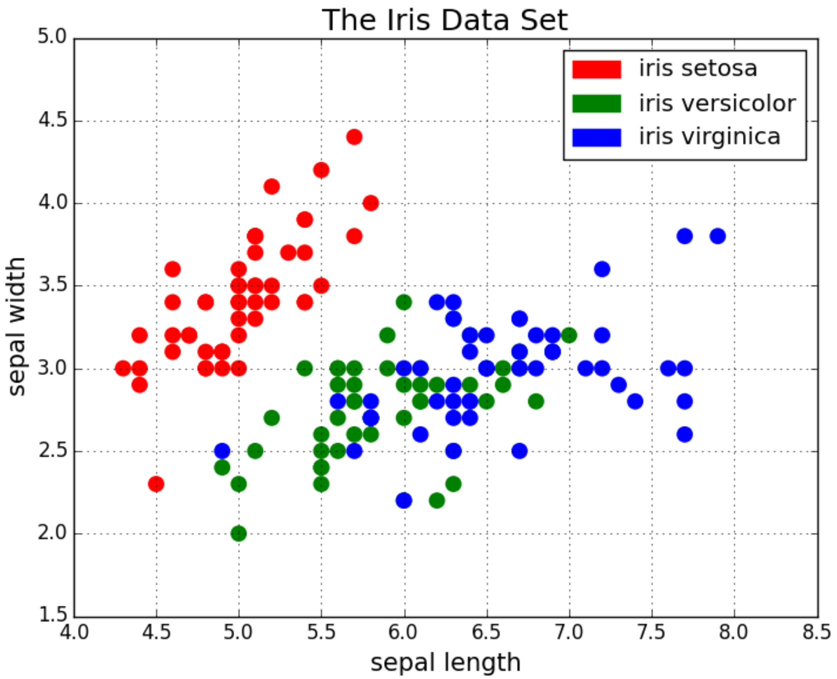
When should we stop training?

- When the perceptron has reached a predefined number of epochs.
- When there are no updates for one full epoch.
  - In that case, the model has **converged**.
  - A model may not converge at all. Why/when?
- When the number of updates per epoch has fallen under a predefined threshold.

Linear classifiers

The feature vectors can be viewed as points in a (high-dimensional) space:

- Two features (x and y axis)
  - In practice, we generally have thousands of features…
- Three classes/labels (red/green/blue)
  - Two-class problems are easier to model

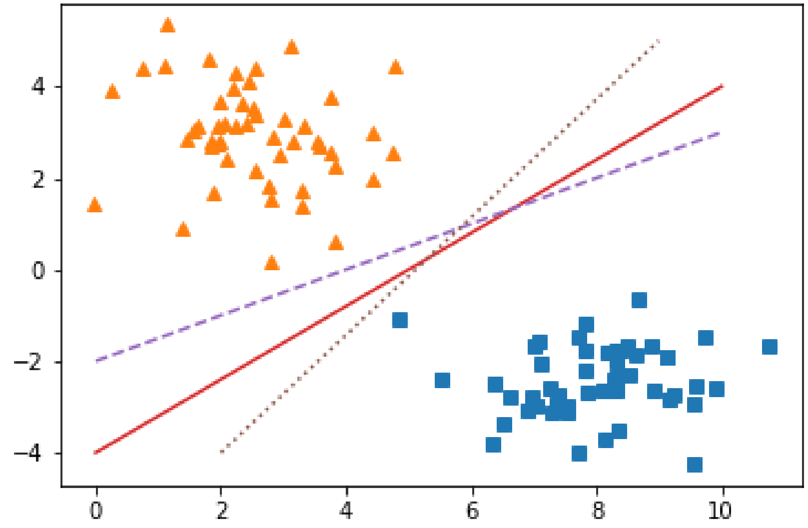


Linear classifiers

The perceptron is a **linear classifier**.

Linear classifiers try to find a **straight line** that separates the instances of the two classes.

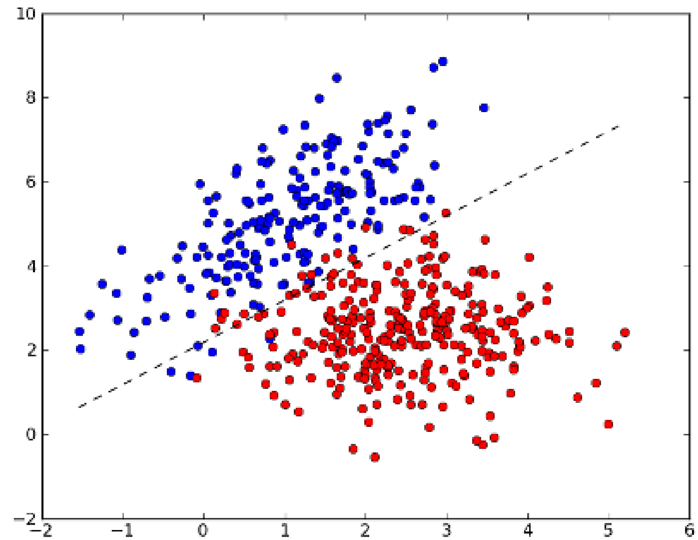
- **Decision boundary**
- What is the “best” such line?
- The perceptron does not give any guarantee on the “goodness” of the line.



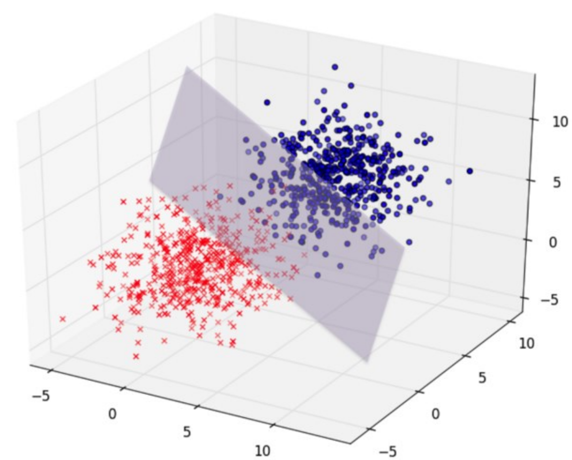
Linear classifiers

The two classes are **linearly separable** if they can be separated by a straight line.

- If the data is not linearly separable, a perceptron will not converge.



In higher-dimensional spaces, the line becomes a **hyper-plane**.

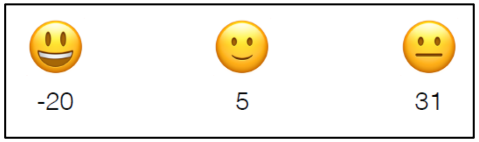


Logistic regression

also known as Maximum Entropy Classifier

Probabilities – useful or not?

When the goal is just prediction, any numeric scoring function is fine.



But an algorithm that offers probabilities over labels is useful if:

- we want to interpret its decisions, i.e. understand why it reached the conclusions it did,
- we want to know how confident the algorithm was or compute statistical tests on its decisions,
- we want the training process to be guided by these confidence values,
- its output is fed as input to some other system.

From scores to probabilities

How can we convert a tuple of numeric scores into a probability distribution?

- Make sure each score is 0 or positive
  - Exponentiation
  - $\exp(x) = e^x = 2.7183^x$
- Make sure that the sum of scores is 1
  - Normalization

This operation is called **softmax**:

$$[x_1 \; x_2 \; \dots \; x_n]$$

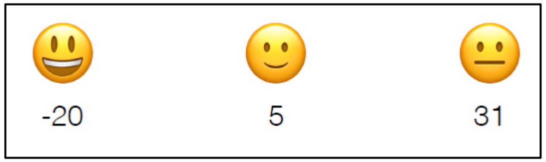
$$\left[\frac{\exp(x_1)}{\sum_{i=1}^{n} \exp(x_i)} \; \frac{\exp(x_2)}{\sum_{i=1}^{n} \exp(x_i)} \; \cdots \; \frac{\exp(x_n)}{\sum_{i=1}^{n} \exp(x_i)} \right]$$

From scores to probabilities

- Exponentiation example:
  - $\exp(-15) = 0.0000003$
  - $\exp(0) = 1$
  - $\exp(15) = 3\ 269\ 017$

$$\boxed{\exp(x) = e^x = 2.7183^x}$$

- Softmax example:
  - $\frac{\exp(-20)}{\exp(-20)+\exp(5)+\exp(31)} \approx 7.1 \cdot 10^{-23}$
  - $\frac{\exp(5)}{\exp(-20)+\exp(5)+\exp(31)} \approx 5.1 \cdot 10^{-12}$
  - $\frac{\exp(31)}{\exp(-20)+\exp(5)+\exp(31)} \approx 0.9999$



Logistic regression

Logistic regression can be viewed a probabilistic variant of the perceptron.

Its prediction scores correspond to **conditional probabilities**:

$$S(x, y) = P(y|x) = \frac{\exp(\boldsymbol{w_y} \cdot \boldsymbol{f_k})}{\sum_{y' \in Y} \exp(\boldsymbol{w_{y'}} \cdot \boldsymbol{f_k})}$$

- Logistic regression can give an indication of how likely it is that the answer is correct.

Perceptron vs logistic regression

	Perceptron	Logistic regression
Feature re-presentation	Vector of binary and/or real-valued feature functions	Vector of binary and/or real-valued feature functions
Scoring function	$S(x_k, y) = \boldsymbol{w_y} \cdot \boldsymbol{f_k}$	$P(y\|x_k) = \frac{\exp(\boldsymbol{w_y} \cdot \boldsymbol{f_k})}{\sum_{y' \in Y} \exp(\boldsymbol{w_{y'}} \cdot \boldsymbol{f_k})}$
Prediction function	$\hat{y} = \underset{y \in Y}{\operatorname{argmax}}\, S(x_k, y)$	$\hat{y} = \underset{y \in Y}{\operatorname{argmax}}\, P(y\|x_k)$
Update rule		

If you are only interested in the predicted class, then the softmax transformation is not necessary…

Example

You are given a logistic regression model for three classes A, B and C.

The current model parameters are $w = \{\boldsymbol{w_A}, \boldsymbol{w_B}, \boldsymbol{w_c}\}$, where $\boldsymbol{w_y}$ is the weight vector for class y:

- $\boldsymbol{w_A} = [\ \ 1.0,\ \ 1.2, -2.0,\ \ 1.5,\ \ 1.0]$
- $\boldsymbol{w_B} = [-2.0,\ \ 3.0,\ \ 1.0,\ \ 0.0, -2.0]$
- $\boldsymbol{w_C} = [\ \ 0.0, -3.0,\ \ 0.0, -2.0,\ \ 5.0]$

You are additionally given an example x_k whose feature vector is $\boldsymbol{f_k} = [0, 1, 0, 1, 1]$

Compute $P(y|x_k)$ for each of the three classes.

Perceptron vs logistic regression

	Perceptron	Logistic regression
Feature re-presentation	Vector of binary and/or real-valued feature functions	Vector of binary and/or real-valued feature functions
Scoring function	$S(x_k, y) = \boldsymbol{w_y} \cdot \boldsymbol{f_k}$	$P(y \mid x_k) = \frac{\exp(\boldsymbol{w_y} \cdot \boldsymbol{f_k})}{\sum_{y' \in Y} \exp(\boldsymbol{w_{y'}} \cdot \boldsymbol{f_k})}$
Prediction function	$\hat{y} = \operatorname{argmax}_{y \in Y} S(x_k, y)$	$\hat{y} = \operatorname{argmax}_{y \in Y} P(y \mid x_k)$
Update rule	$\boldsymbol{w}_y \leftarrow \boldsymbol{w}_y + \boldsymbol{f_k}$ $\boldsymbol{w}_{\hat{y}} \leftarrow \boldsymbol{w}_{\hat{y}} - \boldsymbol{f_k}$	??

Logistic regression update rule

Let y be the gold class:

- $\boldsymbol{w_y} \leftarrow \boldsymbol{w_y} + (1 - P(y|x_k)) \cdot \boldsymbol{f_k}$
- $\boldsymbol{w_z} \leftarrow \boldsymbol{w_z} - P(z|x_k) \cdot \boldsymbol{f_k}$ for all $z \neq y$

- We always update parameters for all classes.
- If the classifier assigns a high probability to an incorrect class z, we strongly update $\boldsymbol{w_z}$.
- If the classifier assigns a high probability to the correct class y, we update $\boldsymbol{w_y}$ only a little.

Logistic regression update rule

The update intensity can be modulated with an additional parameter λ, the **learning rate**:

- $\boldsymbol{w_y} \leftarrow \boldsymbol{w_y} + \lambda \cdot (1 - P(y|x_k)) \cdot \boldsymbol{f_k}$
- $\boldsymbol{w_z} \leftarrow \boldsymbol{w_z} - \lambda \cdot P(z|x_k) \cdot \boldsymbol{f_k}$ for all $z \neq y$

Perceptron vs logistic regression

	Perceptron	Logistic regression
Feature re-presentation	Vector of binary and/or real-valued feature functions	Vector of binary and/or real-valued feature functions
Scoring function	$S(x,y) = \boldsymbol{w_y} \cdot \boldsymbol{f_k}$	$P(y \vert x_k) = \frac{\exp(\boldsymbol{w_y} \cdot \boldsymbol{f_k})}{\sum_{y' \in Y} \exp(\boldsymbol{w_{y'}} \cdot \boldsymbol{f_k})}$
Prediction function	$\hat{y} = \operatorname{argmax}_{y \in Y} S(x,y)$	$\hat{y} = \operatorname{argmax}_{y \in Y} P(y \vert x)$
Update rule	$\boldsymbol{w_y} \leftarrow \boldsymbol{w_y} + \boldsymbol{f_k}$ $\boldsymbol{w_{\hat{y}}} \leftarrow \boldsymbol{w_{\hat{y}}} - \boldsymbol{f_k}$	$\boldsymbol{w_y} \leftarrow \boldsymbol{w_y} + \lambda \cdot (1 - P(y \vert x_k)) \cdot \boldsymbol{f_k}$ $\boldsymbol{w_z} \leftarrow \boldsymbol{w_z} - \lambda \cdot P(z \vert x_k) \cdot \boldsymbol{f_k}$

That's all you need to know to implement a LR classifier.

Perceptron vs logistic regression

1. Are there any conditions where the perceptron and LR update rules produce the same update (assuming $\lambda = 1$)?

2. What happens if $P(y|x) = 1$?

3. What happens if $P(z|x) = 0$ for some $z \neq y$?

When should we stop training?

- After a predefined number of epochs
- When the overall probability of the training data reaches a predefined threshold
  - I.e., minimize training loss
- When the classification performance on the validation set stops improving
  - The validation set can also be used e.g. to find the optimal learning rate.

Some background about logistic regression

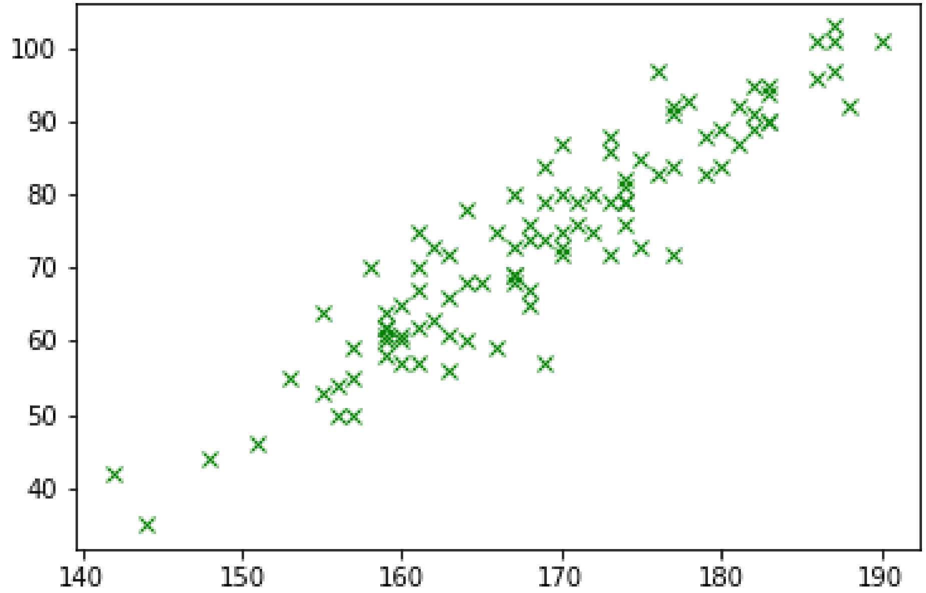
Some background

- Why the name “logistic regression”?
- Different learning strategies
- Regularization

Linear regression

Example: predict the weight (y) from the height (x) of a person

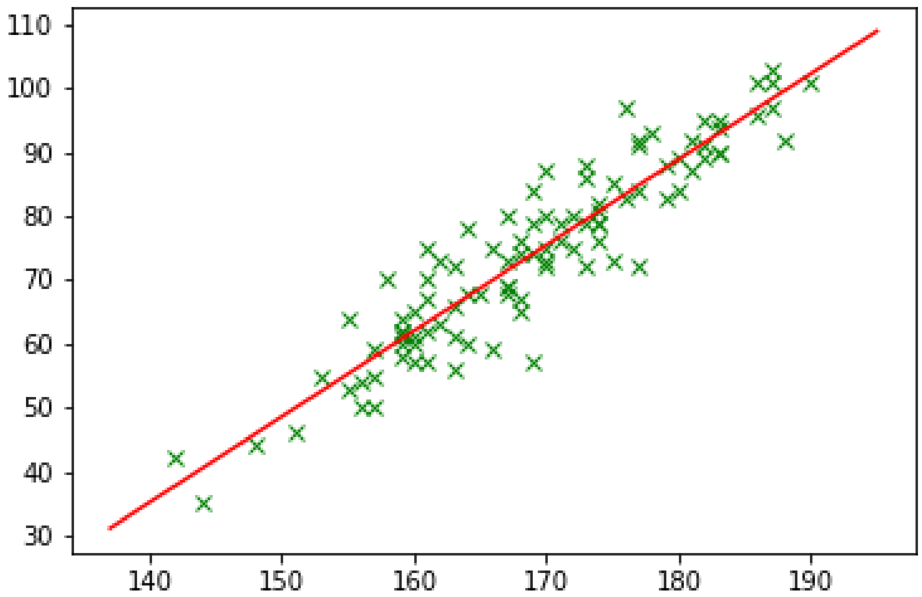
- Both x and y are numeric variables



Linear regression

Method:

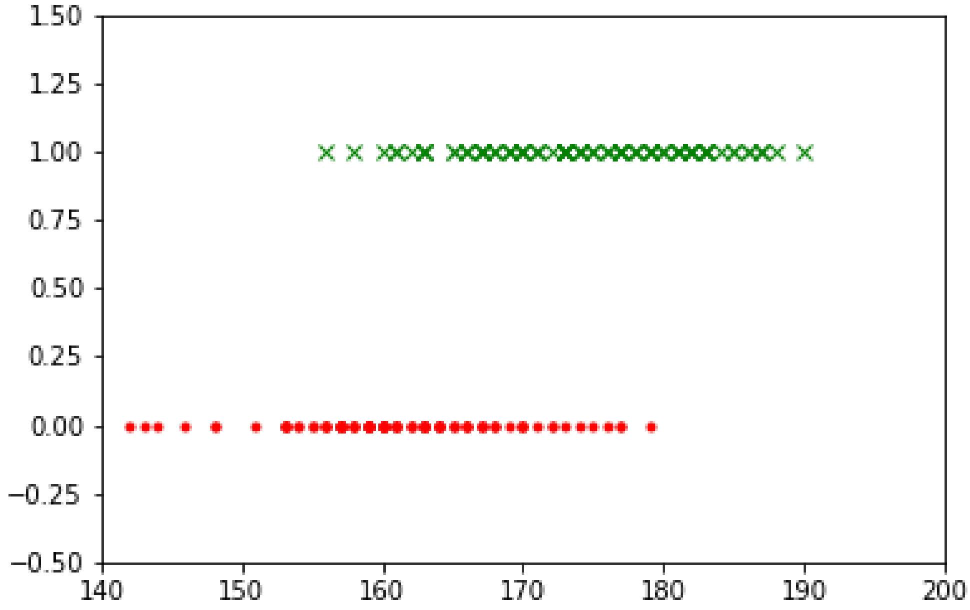
- Fit a straight line to the observed data
- Assume that unseen data are placed on the line



Logistic regression

Another example: predicting gender from height

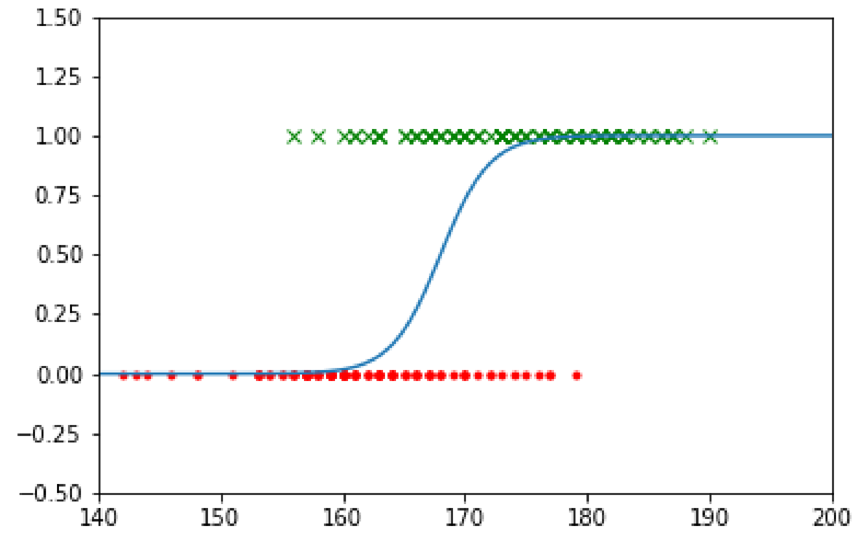
- y is a categorical variable
- Let's assume y is binary (values 0 and 1)



Logistic regression

The optimal regression line is a sigmoid curve:

- This is not a straight line, so the regression cannot be called "linear" here.



The regression line represents the **probability** of predicting gender 1:

- If $P(y = 1|x) \geq 0.5$ predict gender 1 (green)
- If $P(y = 1|x) < 0.5$ predict gender 0 (red)

Logistic regression

Two-class classification:

- Only one weight vector
- Prediction corresponds to the probability of class 1
- Logistic regression scoring function:

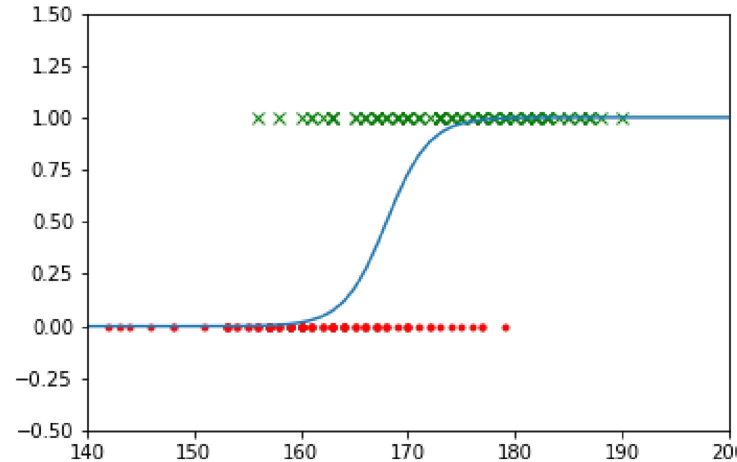
$$P(y = 1 | x_k) = \frac{1}{1 + \exp(\boldsymbol{f}_k \cdot \boldsymbol{w})}$$

- This is called the **sigmoid function** or **logistic function**

Logistic regression

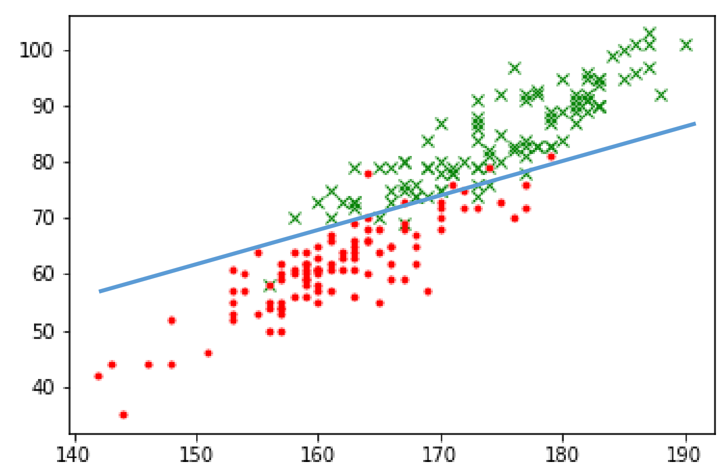
Regression point of view (1 input feature):

- The line is the regression line
- It is sigmoid-shaped
- It represents a probability



Classification point of view (2 input features):

- The line is the decision boundary
- It is a straight line
- It represents the most ambiguous feature values



Learning in logistic regression

During training, we are given labeled training data $\mathcal{D} = \{(\boldsymbol{f_1}, y_1), \ldots, (\boldsymbol{f_n}, y_n)\}$ and our goal is to find the best parameters $\boldsymbol{w}$.

The measure for how well we're doing on dataset $\mathcal{D}$ is the probability of the dataset given the weight vector:

$$\prod_{i=1}^{n} P(y_i | \boldsymbol{f_i}; \boldsymbol{w})$$

By convention, we take the logarithm of this probability. It is called the **objective**:

$$\log\left(\prod_{i=1}^{n} P(y_i | \boldsymbol{f_i}; \boldsymbol{w})\right) = \sum_{i=1}^{n} \log(P(y_i | \boldsymbol{f_i}; \boldsymbol{w}))$$

Our model is good if the value of the objective is large.

Learning in logistic regression

Rather than maximizing an objective, the learning process is usually formalized as minimizing a **loss**.

Logistic regression uses the negative log-likelihood loss or **cross-entropy loss**:

$$L(\boldsymbol{w}) = -\sum_{i=1}^{n} \log(P(y_i|\boldsymbol{x_i};\boldsymbol{w}))$$

- The loss is influenced by the training data and the model parameters. But we assume that the training data is fixed, so we can view the loss as a function of the model parameters.

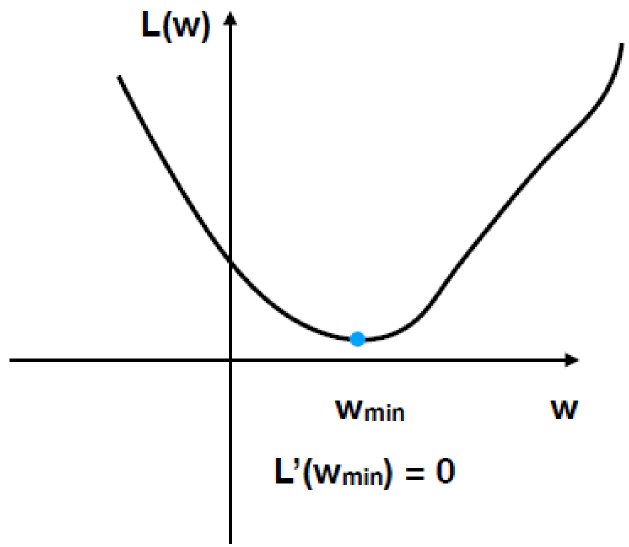
To obtain a better model, we need to adjust the model parameters such that the loss decreases.

Models with 1 parameter

Let us assume that we have a single parameter (feature) w.

Then, we can draw the loss $L(w)$ as a function of w.

- The cross-entropy loss is guaranteed to produce a convex curve with one minimum.
- w is typically initialized at 0, but we want it to become w_{min}.



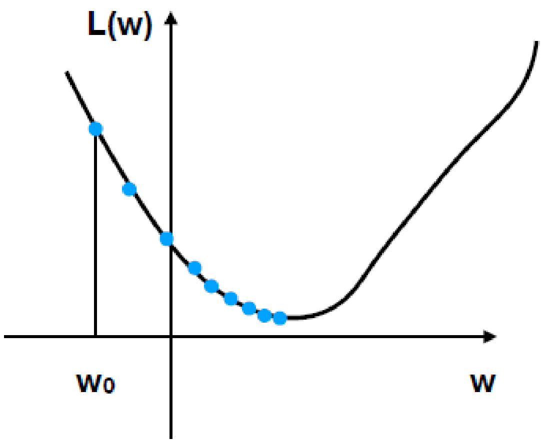
Models with 1 parameter

How do we find the minimum of $L(w)$?

- Take the derivative and set it to zero: $L'(w) = 0$
- Unfortunately, we usually cannot solve the equation $L'(w) = 0$

Instead, we can use an iterative approach:

- Pick a random value for w
- Evaluate $L'(w)$
- If $L'(w) = 0$, we have found the optimal value of w
- If $L'(w) < 0$, we need to increase w
- If $L'(w) > 0$, we need to decrease w



$L'(w) < 0$: the curve goes downwards, i.e. the minimum is on the right
$L'(w) > 0$: the curve goes upwards, i.e. the minimum is on the left

Models with 1 parameter

We can use an iterative approach:

- Pick a random value for w (or set $w = 0$)
- Evaluate $L'(w)$
- Update $w \leftarrow w - \lambda \cdot L'(w)$
- Repeat until $L'(w)$ becomes very small

If $L'(w) = -1$, increase w by λ
If $L'(w) = +1$, decrease w by λ
If $L'(w) = 0$, w doesn't change

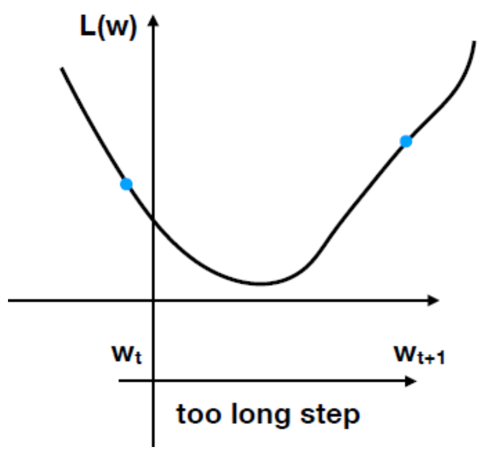
This procedure is called **gradient descent**.

- For $n > 1$ parameters, the derivative $L'(w)$ is replaced with its n-dimensional equivalent, the gradient $\nabla L(\mathbf{w})$

Learning rate

The learning rate λ governs the step size in gradient descent.

- If λ is too large, we may end up taking too large steps and may miss the optimum of the loss function.
- Here we take a step in the right direction but the step is too long. We end up getting a higher value for the loss than we had before!



Gradient descent

Generic update rule: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \cdot \nabla L(\boldsymbol{w})$

If you...

- take the cross-entropy loss function,
- substitute $P(y_i|\boldsymbol{x_i};\boldsymbol{w})$ by the softmax,
- take its gradient,
- and plug it into the generic update rule,

There is a demonstration of this process for the two-class LR in J&M 5.10.

then you should end up with the update rule specified in the beginning:

- $\boldsymbol{w_y} \leftarrow \boldsymbol{w_y} + \lambda \cdot (1 - P(y|x)) \cdot \boldsymbol{f_x}$
- $\boldsymbol{w_z} \leftarrow \boldsymbol{w_z} - \lambda \cdot P(z|x) \cdot \boldsymbol{f_x}$ for all $z \neq y$

Different learning strategies

1. Batch learning:

- Calculate the loss for the whole training set:

$$L(\boldsymbol{w}) = -\sum_{i=1}^{n} \log(P(y_i|\boldsymbol{x_i};\boldsymbol{w}))$$

- Make one move in the direction of the gradient
- Repeat

- Slow and inefficient, only makes one update per epoch!

Different learning strategies

2. Stochastic gradient descent:

- Randomly pick one item from the training data
- Calculate the loss for this item
- Move in the direction of the gradient for this item

- Idea: the loss of the entire training set can be approximated by the loss of one randomly chosen example.
- Faster, but unstable.

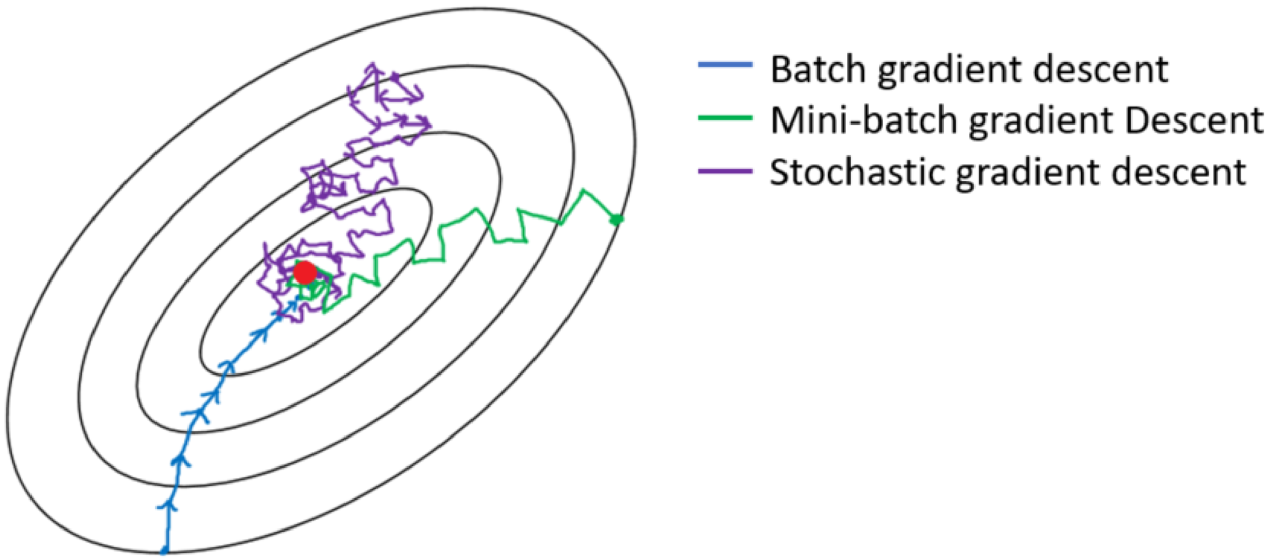
Different learning strategies

3. Minibatch training:

- Sample a small number of instances from the training data
- Calculate the loss for this subset
- Make one move in the direction of this gradient

- Good compromise between speed and stability
- Standard approach used with neural networks

Different learning strategies



https://suniljangirblog.wordpress.com/2018/12/13/variants-of-gradient-descent/

Regularization

Logistic regression is prone to overfitting to the training data.

Regularization reduces overfitting by penalizing large weight values.

- No single feature/weight should override the others.

- L2 regularization: $R(w) = \sum_0^n w_i^2$
- L1 regularization: $R(w) = \sum_0^n |w_i|$
- Can be specified with the *penalty* and *C* parameters in Scikit-Learn.

Readings

- Jurafsky & Martin, chapter 5
  - Note: this chapter doesn’t cover the perceptron.