Linear Classifiers

André Martins

Lisbon Machine Learning School, July 8, 2021
Why Linear Classifiers?

It’s 2021 and everybody uses neural networks. Why a lecture on linear classifiers?

• The underlying machine learning concepts are the same
• The theory (statistics and optimization) are much better understood
• Linear classifiers are still widely used (and very effective when data is scarce)
• Linear classifiers are a component of neural networks.
Linear Classifiers and Neural Networks

Linear Classifier

Linear Classifier

Linear Classifier

Dog

Cat

Linear Classifier
Linear Classifiers and Neural Networks

Handcrafted Features

Linear Classifier

Dog
Cat
Today's Roadmap

- Linear regression
- Binary and multi-class classification
- Linear classifiers: perceptron, naive Bayes, logistic regression, SVMs
- Softmax and sparsemax
- Regularization and optimization, stochastic gradient descent
- Similarity-based classifiers and kernels.
Example Tasks

**Binary:** given an e-mail: is it spam or not-spam?

**Multi-class:** given a news article, determine its topic (politics, sports, etc.)
Outline

1 Data and Feature Representation

2 Regression

3 Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines

4 Regularization

5 Non-Linear Classifiers
Some of the following slides are adapted from Ryan McDonald.
Let’s Start Simple

• Example 1 – sequence: ★ ◊ ○;  
  label: −1
• Example 2 – sequence: ★ ♥ △;  
  label: −1
• Example 3 – sequence: ★ △ ♠;  
  label: +1
• Example 4 – sequence: ◊ △ ○;  
  label: +1

Why can we do this?

André Martins (IST)

Linear Classifiers

LxMLS 2021 8 / 158
Let’s Start Simple

- Example 1 – sequence: ★ ◊ ◯; label: −1
- Example 2 – sequence: ★ ♦ △; label: −1
- Example 3 – sequence: ★ △ ♠; label: +1
- Example 4 – sequence: ◊ △ ◯; label: +1

- New sequence: ★ ◊ ◯; label ?
Let’s Start Simple

• Example 1 – sequence: ⋆⋄○; label: −1
• Example 2 – sequence: ⋆♡△; label: −1
• Example 3 – sequence: ⋆△♠; label: +1
• Example 4 – sequence: ⋇△○; label: +1

• New sequence: ⋆⋄○; label −1
• New sequence: ⋆⋄♡; label ?
Let’s Start Simple

• Example 1 – sequence: ★ ◇ ○; label: −1
• Example 2 – sequence: ★ ♠ △; label: −1
• Example 3 – sequence: ★ △ ♠; label: +1
• Example 4 – sequence: ◇ △ ○; label: +1

• New sequence: ★ ◇ ○; label −1
• New sequence: ★ ◇ ♠; label −1
• New sequence: ★ △ ○; label ?
Let’s Start Simple

- Example 1 – sequence: \( \star \diamond \circ; \) label: \(-1\)
- Example 2 – sequence: \( \star \heartsuit \triangle; \) label: \(-1\)
- Example 3 – sequence: \( \star \triangle \spade; \) label: \(+1\)
- Example 4 – sequence: \( \diamond \triangle \circ; \) label: \(+1\)

- New sequence: \( \star \diamond \circ; \) label \(-1\)
- New sequence: \( \star \diamond \heartsuit; \) label \(-1\)
- New sequence: \( \star \triangle \circ; \) label ?

Why can we do this?
Let’s Start Simple: Machine Learning

- Example 1 – sequence: ⋄ ◦ ⋆; label: −1
- Example 2 – sequence: ⋆ ♥ △; label: −1
- Example 3 – sequence: ⋆ △ ♠; label: +1
- Example 4 – sequence: ◆ △ ◆; label: +1

- New sequence: ⋆ ◆ ♥; label −1

\[
\begin{align*}
P(-1|⋆) &= \frac{\text{count}(⋆ \text{ and } -1)}{\text{count}(⋆)} = \frac{2}{3} = 0.67 \\
P(-1|◆) &= \frac{\text{count}(◆ \text{ and } -1)}{\text{count}(◆)} = \frac{1}{2} = 0.5 \\
P(-1|♥) &= \frac{\text{count}(♥ \text{ and } -1)}{\text{count}(♥)} = \frac{1}{1} = 1.0
\end{align*}
\]

\[
\begin{align*}
P(+1|⋆) &= \frac{\text{count}(⋆ \text{ and } +1)}{\text{count}(⋆)} = \frac{1}{3} = 0.33 \\
P(+1|◆) &= \frac{\text{count}(◆ \text{ and } +1)}{\text{count}(◆)} = \frac{1}{2} = 0.5 \\
P(+1|♥) &= \frac{\text{count}(♥ \text{ and } +1)}{\text{count}(♥)} = \frac{0}{1} = 0.0
\end{align*}
\]
Let’s Start Simple: Machine Learning

• Example 1 – sequence: $\star \lozenge \circ$; label: $-1$
• Example 2 – sequence: $\star \heartsuit \triangle$; label: $-1$
• Example 3 – sequence: $\star \triangle \spadesuit$; label: $+1$
• Example 4 – sequence: $\lozenge \triangle \circ$; label: $+1$

• New sequence: $\star \triangle \circ$; label ?

$\begin{align*}
P(-1|\star) &= \frac{\text{count}(\star \text{ and } -1)}{\text{count}(\star)} = \frac{2}{3} = 0.67 \text{ vs. } P(+1|\star) = \frac{\text{count}(\star \text{ and } +1)}{\text{count}(\star)} = \frac{1}{3} = 0.33 \\
P(-1|\triangle) &= \frac{\text{count}(\triangle \text{ and } -1)}{\text{count}(\triangle)} = \frac{1}{3} = 0.33 \text{ vs. } P(+1|\triangle) = \frac{\text{count}(\triangle \text{ and } +1)}{\text{count}(\triangle)} = \frac{2}{3} = 0.67 \\
P(-1|\circ) &= \frac{\text{count}(\circ \text{ and } -1)}{\text{count}(\circ)} = \frac{1}{2} = 0.5 \text{ vs. } P(+1|\circ) = \frac{\text{count}(\circ \text{ and } +1)}{\text{count}(\circ)} = \frac{1}{2} = 0.5
\end{align*}$
1. Define a model/distribution of interest
2. Make some assumptions if needed
3. Fit the model to the data
1. Define a model/distribution of interest
2. Make some assumptions if needed
3. Fit the model to the data

- Model: $P(\text{label} | \text{sequence}) = P(\text{label} | \text{symbol}_1, \ldots, \text{symbol}_n)$
  - Prediction for new sequence = $\arg\max_{\text{label}} P(\text{label} | \text{sequence})$

- Assumption (naive Bayes—more later):
  
  $$P(\text{symbol}_1, \ldots, \text{symbol}_n | \text{label}) = \prod_{i=1}^{n} P(\text{symbol}_i | \text{label})$$

- Fit the model to the data: count!! (simple probabilistic modeling)
Some Notation: Inputs and Outputs

• Input $x \in X$
  • e.g., a news article, a sentence, an image, ...

• Output $y \in Y$
  • e.g., spam/not spam, a topic, a parse tree, an image segmentation

• Input/Output pair: $(x, y) \in X \times Y$
  • e.g., a news article together with a topic
  • e.g., a sentence together with a parse tree
  • e.g., an image partitioned into segmentation regions
Supervised Machine Learning

- We are given a labeled dataset of input/output pairs:

\[ D = \{ (x_n, y_n) \}_{n=1}^{N} \subseteq X \times Y \]

- **Goal:** use it to learn a predictor \( h : X \rightarrow Y \) that generalizes well to arbitrary inputs.

- At test time, given \( x \in X \), we predict

\[ \hat{y} = h(x). \]

- Hopefully, \( \hat{y} \approx y \) most of the time.
Things can go by different names depending on what $y$ is...
Deals with *continuous* output variables:

- **Regression**: \( y = \mathbb{R} \)
  
  - e.g., given a news article, how much time a user will spend reading it?

- **Multivariate regression**: \( y = \mathbb{R}^K \)
  
  - e.g., predict the X-Y coordinates in an image where the user will click
Deals with *discrete* output variables:

- **Binary classification:** $y = \{\pm 1\}$
  - e.g., spam detection

- **Multi-class classification:** $y = \{1, 2, \ldots, K\}$
  - e.g., topic classification

- **Structured classification:** $y$ exponentially large and structured
  - e.g., machine translation, caption generation, image segmentation

See Xavier Carreras' lecture later at LxMLS!
Deals with **discrete** output variables:

- **Binary classification**: $y = \{\pm 1\}$
  - e.g., spam detection

- **Multi-class classification**: $y = \{1, 2, \ldots, K\}$
  - e.g., topic classification

- **Structured classification**: $y$ exponentially large and structured
  - e.g., machine translation, caption generation, image segmentation
  - See Xavier Carreras’ lecture later at LxMLS!
Deals with **discrete** output variables:

- **Binary classification:** $y = \{ \pm 1 \}$
  - e.g., spam detection

- **Multi-class classification:** $y = \{1, 2, \ldots, K\}$
  - e.g., topic classification

- **Structured classification:** $y$ exponentially large and structured
  - e.g., machine translation, caption generation, image segmentation
  - See Xavier Carreras’ lecture later at LxMLS!

*Today we’ll focus mostly on multi-class classification.*
Sometimes reductions are convenient:

- logistic regression reduces classification to regression
- one-vs-all reduces multi-class to binary
- greedy search reduces structured classification to multi-class

... but other times it’s better to tackle the problem in its native form.

More later!
Feature engineering is an important step in linear classifiers:

- Bag-of-words features for text, also lemmas, parts-of-speech, ...
- SIFT features and wavelet representations in computer vision
- Other categorical, Boolean, and continuous features
We need to represent information about \( x \)

**Typical approach:** define a feature map \( \phi : \mathcal{X} \rightarrow \mathbb{R}^D \)

- \( \phi(x) \) is a high dimensional feature vector

We can use feature vectors to encapsulate **Boolean, categorical, and continuous** features

- e.g., categorical features can be reduced to a range of one-hot binary values.
Example: Continuous Features

Handcrafted Features

Linear Classifier
Classical NLP pipelines consist of stacking together several linear classifiers. Each classifier’s predictions are used to handcraft features for other classifiers.

Examples of features:

- **Word occurrences**: binary feature denoting if a word occurs in or not in a document.
- **Word counts**: real-valued feature counting how many times a word occurs.
- **POS tags**: adjective counts for sentiment analysis.
- **Spell checker**: misspellings counts for spam detection.
Example: Translation Quality Estimation

Translate

does machine translation work?

Le travail de traduction automatique?
Example: Translation Quality Estimation

Wrong translation!

Google
Translate

does machine translation work?

Le travail de traduction automatique?
Example: Translation Quality Estimation

Wrong translation!

Google

Translate

English  Spanish  French  Detect language

French  Spanish  Portuguese  Translate

Translate

does machine translation work?

Le travail de traduction automatique?

Goal: estimate the quality of a translation on the fly (without a reference)!
Hand-crafted features:

- no of tokens in the source/target segment
- LM probability of source/target segment and their ratio
- % of source 1–3-grams observed in 4 frequency quartiles of source corpus
- average no of translations per source word
- ratio of brackets and punctuation symbols in source & target segments
- ratio of numbers, content/non-content words in source & target segments
- ratio of nouns/verbs/etc in the source & target segments
- % of dependency relations b/w constituents in source & target segments
- diff in depth of the syntactic trees of source & target segments
- diff in no of PP/NP/VP/ADJP/ADVP/CONJP in source & target
- diff in no of person/location/organization entities in source & target
- features and global score of the SMT system
- number of distinct hypotheses in the n-best list
- 1–3-gram LM probabilities using translations in the n-best to train the LM
- average size of the target phrases
- proportion of pruned search graph nodes;
- proportion of recomposed graph nodes.
Feature engineering is a black art and can be very time-consuming

But it’s a good way of encoding prior knowledge, and it is still widely used in practice (in particular with “small data”)

One alternative to feature engineering: representation learning
Feature engineering is a black art and can be very time-consuming

But it's a good way of encoding prior knowledge, and it is still widely used in practice (in particular with “small data”)

One alternative to feature engineering: representation learning

Bhiksha will talk about this tomorrow!
Outline

1 Data and Feature Representation

2 Regression

3 Classification
   Perceptron
   Naive Bayes
   Logistic Regression
   Support Vector Machines

4 Regularization

5 Non-Linear Classifiers
Output space $y$ is continuous

Example: given an article, how much time a user spends reading it?

- $x$ is number of words of the article
- $y$ is the reading time (minutes)

How to define a model that predicts $\hat{y}$ from $x$?
• First take: assume $\hat{y} = wx + b$
• Model parameters: $w$ and $b$
• Given training data $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$, how to estimate $w$ and $b$?

Least squares method: fit $w$ and $b$ on the training set by minimizing

$$\sum_{n=1}^{N}(y_n - (wx_n + b))^2$$
• First take: assume $\hat{y} = wx + b$

• Model parameters: $w$ and $b$

• Given training data $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$, how to estimate $w$ and $b$?

**Least squares method:** fit $w$ and $b$ on the training set by minimizing

$$\sum_{n=1}^N (y_n - (wx_n + b))^2$$
Often a linear dependency of $\hat{y}$ on $x$ is a poor assumption

Second take: assume $\hat{y} = w \cdot \phi(x)$, where $\phi(x)$ is a feature vector

- e.g. $\phi(x) = [1, x, x^2, \ldots, x^D]$ (polynomial features degree $\leq D$)
- the bias term $b$ is captured by the constant feature $\phi_0(x) = 1$

Fit $w$ by minimizing $\sum_n (y_n - (w \cdot \phi(x_n)))^2$

- Closed form solution:

\[
    w = (X^\top X)^{-1} X^\top y, \quad \text{with} \quad X = \begin{bmatrix} \vdots \phi(x_n)^\top \vdots \end{bmatrix}, \quad y = \begin{bmatrix} \vdots y_n \vdots \end{bmatrix}.
\]

Still called linear regression – linearity w.r.t. the model parameters $w$. 
Linear Regression \((D = 1)\)
Linear Regression ($D = 2$)
Linear regression with the least squares method corresponds to a loss function

\[ L(y, \hat{y}) = \frac{1}{2} (y - \hat{y})^2, \quad \text{where} \quad \hat{y} = \mathbf{w} \cdot \phi(x). \]

The model is fit to the training data by minimizing this loss function. This is called the **squared loss**.

More later.
The least squares method has a probabilistic interpretation. Assume the data is generated stochastically as

$$y = w^* \cdot \phi(x) + n$$

where $n \sim \mathcal{N}(0, \sigma^2)$ is Gaussian noise (with $\sigma$ fixed), and $w^*$ are the “true” model parameters.

That is, $y \sim \mathcal{N}(w^* \cdot \phi(x), \sigma^2)$.

Then $w$ given by least squares is the maximum likelihood estimate under this model.
Recall \( \mathcal{N}(y; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} \exp \left( -\frac{(y-\mu)^2}{2\sigma^2} \right) \).

\[
\hat{w}_{\text{MLE}} = \arg \max_w \prod_{n=1}^{N} P(y_n \mid x_n; w)
\]

\[
= \arg \max_w \sum_{n=1}^{N} \log P(y_n \mid x_n; w)
\]

\[
= \arg \max_w \sum_{n=1}^{N} -\frac{(y_n - w \cdot \phi(x_n))^2}{2\sigma^2} - \log(\sqrt{2\pi\sigma})
\]

\[
= \arg \min_w \sum_{n=1}^{N} (y_n - w \cdot \phi(x_n))^2
\]

Thus, linear regression with the squared loss = MLE under Gaussian noise.
Other Regression Losses

Squared loss: \( L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2. \)

Absolute error loss: \( L(y, \hat{y}) = |y - \hat{y}|. \)

Huber loss: \( L(y, \hat{y}) = \begin{cases} 
\frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq 1 \\
|y - \hat{y}| - \frac{1}{2} & \text{if } |y - \hat{y}| \geq 1. 
\end{cases} \)
We saw earlier an example of underfitting.

However, if the model is too complex (too many parameters) and the data is scarce, we run the risk of overfitting:

To avoid overfitting, we need regularization (more later).
Assuming we have a prior distribution on $w$, $w \sim \mathcal{N}(0, \sigma_w^2 I)$

A criterion to estimate $w^*$ is maximum a posteriori (MAP):

$$\hat{w}_{\text{MAP}} = \arg \max_w P(w) \prod_{n=1}^{N} P(y_n | x_n; w)$$

$$= \arg \max_w \log P(w) + \sum_{n=1}^{N} \log P(y_n | x_n; w)$$

$$= \arg \max_w \frac{||w||^2}{2\sigma_w^2} - \sum_{n=1}^{N} \frac{(y_n - w \cdot \phi(x_n))^2}{2\sigma^2} + \text{constant}$$

$$= \arg \min_w \frac{\lambda||w||^2}{2} + \sum_{n=1}^{N} (y_n - w \cdot \phi(x_n))^2$$

Thus, $\ell_2$-regularization is equivalent to MAP with a Gaussian prior.
1 Data and Feature Representation

2 Regression

3 Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines

4 Regularization

5 Non-Linear Classifiers
Before covering multi-class classification, we address the simpler case of binary classification.

Output space $Y = \{-1, +1\}$

Example: Given a news article, is it true or fake?

- $x$ is the news article, represented a feature vector $\phi(x)$
- $y$ can be either true (+1) or fake (−1)

How to define a model to predict $\hat{y}$ from $x$?
Defined by $\hat{y} = \text{sign}(w \cdot \phi(x) + b) = \begin{cases} +1 & \text{if } w \cdot \phi(x) + b \geq 0 \\ -1 & \text{if } w \cdot \phi(x) + b < 0 \end{cases}$.

Intuitively, $w \cdot \phi(x) + b$ is a “score” for the positive class: if positive, predict +1; if negative, predict −1.

Difference from regression: the sign function converts from continuous to binary.

The decision boundary is an hyperplane defined by the model parameters $w$ and $b$.

Also called a “hyperplane classifier.”
\((w, b)\) is an hyperplane that splits the space into two half spaces:

How to learn this hyperplane from the training data \(\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N}\)?
A dataset $\mathcal{D}$ is **linearly separable** if there exists $(\mathbf{w}, b)$ such that classification is perfect.

We next present an algorithm that finds such an hyperplane if it exists!
It is common to present linear classifiers without the bias term $b$:

$$\hat{y} = \text{sign}(w \cdot \phi(x) + b)$$

In this case, the decision boundary is a hyperplane that passes through the origin.

We can always do this without loss of generality:

- Add a constant feature to $\phi(x)$: $\phi_0(x) = 1$
- Then the corresponding weight $w_0$ replaces the bias term $b$
Outline

1. Data and Feature Representation
2. Regression
3. Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines
4. Regularization
5. Non-Linear Classifiers
Perceptron (Rosenblatt, 1958)

- Invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt
- Implemented in custom-built hardware as the “Mark 1 perceptron,” designed for image recognition
- 400 photocells, randomly connected to the “neurons.” Weights were encoded in potentiometers
- Weight updates during learning were performed by electric motors.

(Extracted from Wikipedia)
NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) — The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo — the Weather Bureau’s $2,000,000 “704” computer — learned to differentiate between right and left after fifty attempts in the Navy’s demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism “capable of receiving, recognizing and identifying its surroundings without any human training or control.”

The “brain” is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

1958 New York Times...

In today’s demonstration, the “704” was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a “Q” for the left squares and “O” for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a “self-induced change in the wiring diagram.”

The first Perceptron will have about 1,000 electronic “association cells” receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.
NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) — The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau’s $2,000,000 “704” computer—learned to differentiate between right and left after fifty attempts in the Navy’s demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism “capable of receiving, recognizing and identifying its surroundings without any human training or control.”

The “brain” is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

1958 New York Times...

In today’s demonstration, the “704” was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a “Q” for the left squares and “O” for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a “self-induced change in the wiring diagram.”

The first Perceptron will have about 1,000 electronic “association cells” receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.
Online algorithm: process one data point at each round

1. Take $x_i$; apply the current model to make a prediction for it
2. If prediction is correct, do nothing
3. Else, correct model $w$ by adding/subtracting feature vector $\phi(x_i)$

For simplicity, omit the bias $b$: assume a constant feature $\phi_0(x) = 1$ as explained earlier.
Perceptron Algorithm

**input:** labeled data $\mathcal{D}$
initialize $w^{(0)} = 0$
initialize $k = 0$ (number of mistakes)

**repeat**
- get new training example $(x_i, y_i)$
- predict $\hat{y}_i = \text{sign}(w^{(k)} \cdot \phi(x_i))$
- **if** $\hat{y}_i \neq y_i$ **then**
  - update $w^{(k+1)} = w^{(k)} + y_i \phi(x_i)$
  - increment $k$
- **end if**

**until** maximum number of epochs

**output:** model weights $w^{(k)}$
Perceptron’s Mistake Bound

A couple definitions:

- the training data is **linearly separable** with margin $\gamma > 0$ iff there is a weight vector $u$ with $\|u\| = 1$ such that
  \[ y_i \ u \cdot \phi(x_i) \geq \gamma, \quad \forall i. \]

- radius of the data: $R = \max_i \| \phi(x_i) \|$. 

Theorem (Novikoff (1962))
The perceptron algorithm is guaranteed to find a separating hyperplane after at most $R^2 \gamma^2$ mistakes.
A couple definitions:

- the training data is **linearly separable** with margin $\gamma > 0$ iff there is a weight vector $\mathbf{u}$ with $\|\mathbf{u}\| = 1$ such that

$$y_i \mathbf{u} \cdot \phi(x_i) \geq \gamma, \quad \forall i.$$ 

- radius of the data: $R = \max_i \|\phi(x_i)\|$.  

Then we have the following bound of the number of mistakes:

**Theorem (Novikoff (1962))**

*The perceptron algorithm is guaranteed to find a separating hyperplane after at most $\frac{R^2}{\gamma^2}$ mistakes.*
Recall that $w^{(k+1)} = w^{(k)} + y_i \phi(x_i)$.

- **Lower bound on $\|w^{(k+1)}\|$**:

  $$u \cdot w^{(k+1)} = u \cdot w^{(k)} + y_i u \cdot \phi(x_i)$$
  $$\geq u \cdot w^{(k)} + \gamma$$
  $$\geq k \gamma.$$

  Hence $\|w^{(k+1)}\| = \|u\| \cdot \|w^{(k+1)}\| \geq u \cdot w^{(k+1)} \geq k \gamma$ (from CSI).
Recall that $w^{(k+1)} = w^{(k)} + y_i \phi(x_i)$.

• **Lower bound on** $\|w^{(k+1)}\|$

\[
\begin{align*}
    u \cdot w^{(k+1)} &= u \cdot w^{(k)} + y_i u \cdot \phi(x_i) \\
    &\geq u \cdot w^{(k)} + \gamma \\
    &\geq k\gamma.
\end{align*}
\]

Hence $\|w^{(k+1)}\| = \|u\| \cdot \|w^{(k+1)}\| \geq u \cdot w^{(k+1)} \geq k\gamma$ (from CSI).

• **Upper bound on** $\|w^{(k+1)}\|$:

\[
\begin{align*}
    \|w^{(k+1)}\|^2 &= \|w^{(k)}\|^2 + \|\phi(x_i)\|^2 + 2y_i w^{(k)} \cdot \phi(x_i) \\
    &\leq \|w^{(k)}\|^2 + R^2 \\
    &\leq kR^2.
\end{align*}
\]

Equating both sides, we get $(k\gamma)^2 \leq kR^2 \Rightarrow k \leq R^2 / \gamma^2$ (QED).
What a Simple Perceptron Can and Can’t Do

- Remember: the decision boundary is linear (linear classifier)
- It can solve linearly separable problems (OR, AND)
What a Simple Perceptron Can and Can’t Do

• ... but it can’t solve non-linearly separable problems such as simple XOR (unless input is transformed into a better representation):

This result is often attributed to Minsky and Papert (1969) but was known well before.
Limitations of the Perceptron

Minsky and Papert (1969):
- Shows limitations of multi-layer perceptrons and fostered an “AI winter” period.

More tomorrow at Bhiksha’s lecture!
Let’s now assume a multi-class classification problem, with $|\mathcal{Y}| \geq 2$ labels (classes).
One strategy for multi-class classification is to train one binary classifier per label (using all the other classes as negative examples) and pick the class with the highest score (one-vs-all).

Another strategy is to train pairwise classifiers and to use majority voting (one-vs-one).

Here, we’ll consider classifiers that tackle the multiple classes directly.
Multi-Class Linear Classifiers

- Parametrized by a **weight matrix** \( W \in \mathbb{R}^{|Y| \times D} \) (one weight per feature/label pair) and a **bias vector** \( b \in \mathbb{R}^{|Y|} \):

  \[
  W = \begin{bmatrix}
  \vdots \\
  w_y^T \\
  \vdots 
  \end{bmatrix}, \quad b = \begin{bmatrix}
  \vdots \\
  b_y \\
  \vdots 
  \end{bmatrix}.
  \]

- Equivalently, \(|Y|\) weight vectors \( w_y \in \mathbb{R}^D \) and scalars \( b_y \in \mathbb{R} \)

- The score (or probability) of a particular label is based on a **linear** combination of features and their weights

- Predict the \( \hat{y} \) which maximizes this score:

  \[
  \hat{y} = \arg \max_{y \in Y} w_y \cdot \phi(x) + b_y.
  \]
Geometrically, \((\mathbf{W}, b)\) split the feature space into regions delimited by hyperplanes.
Commonly Used Notation in Neural Networks

\[
\hat{y} = \arg\max (W \phi(x) + b), \quad W = \begin{bmatrix} \vdots \\ w_y^\top \\ \vdots \end{bmatrix}, \quad b = \begin{bmatrix} \vdots \\ b_y \\ \vdots \end{bmatrix}.
\]
With two classes ($\mathcal{Y} = \{\pm 1\}$), this formulation recovers the binary classifier presented earlier:

$$\hat{y} = \arg \max_{y \in \{\pm 1\}} w_y \cdot \phi(x) + b_y$$
Multi-Class Recovers Binary

With two classes ($\mathcal{Y} = \{\pm 1\}$), this formulation recovers the binary classifier presented earlier:

$$\hat{y} = \arg \max_{y \in \{\pm 1\}} w_y \cdot \phi(x) + b_y$$

$$= \begin{cases} 
+1 & \text{if } w_{+1} \cdot \phi(x) + b_{+1} > w_{-1} \cdot \phi(x) + b_{-1} \\
-1 & \text{otherwise}
\end{cases}$$
With two classes ($\mathcal{Y} = \{ \pm 1 \}$), this formulation recovers the binary classifier presented earlier:

$$
\hat{y} = \arg \max_{y \in \{ \pm 1 \}} w_y \cdot \phi(x) + b_y
$$

$$
= \begin{cases} 
+1 & \text{if } w_{+1} \cdot \phi(x) + b_{+1} > w_{-1} \cdot \phi(x) + b_{-1} \\
-1 & \text{otherwise}
\end{cases}
$$

$$
= \text{sign}( (w_{+1} - w_{-1}) \cdot \phi(x) + (b_{+1} - b_{-1}) ).
$$
With two classes \( \mathcal{Y} = \{ \pm 1 \} \), this formulation recovers the binary classifier presented earlier:

\[
\hat{y} = \arg \max_{y \in \{ \pm 1 \}} \ w_y \cdot \phi(x) + b_y
\]

\[
= \begin{cases} 
+1 & \text{if } w_{+1} \cdot \phi(x) + b_{+1} > w_{-1} \cdot \phi(x) + b_{-1} \\
-1 & \text{otherwise}
\end{cases}
\]

\[
= \text{sign}( (w_{+1} - w_{-1}) \cdot \phi(x) + (b_{+1} - b_{-1}) ).
\]

That is: only half of the parameters are needed.
Linear Classifiers (Binary vs Multi-Class)

- Prediction rule:

\[ \hat{y} = h(x) = \arg \max_{y \in \mathcal{Y}} w_y \cdot \phi(x) \]

- The decision boundary is defined by the intersection of half spaces
- In the binary case ($|\mathcal{Y}| = 2$) this corresponds to a hyperplane classifier
Again, it is common to omit the bias vector $b$:

$$\hat{y} = \arg \max_{y \in Y} w_y \cdot \phi(x) + b_y$$

Like before, this can be done without loss of generality, by assuming a constant feature $\phi_0(x) = 1$

The first column of $W$ replaces the bias vector.

We assume this for simplicity.
The perceptron algorithm also works for the multi-class case!

It has a similar mistake bound: if the data is separable, it’s guaranteed to find separating hyperplanes!
input: labeled data $\mathcal{D}$
initialize $W^{(0)} = 0$
initialize $k = 0$ (number of mistakes)
repeat
  get new training example $(x_i, y_i)$
  predict $\hat{y}_i = \arg\max_{y \in Y} w_y^{(k)} \cdot \phi(x_i)$
  if $\hat{y}_i \neq y_i$ then
    update $w_{y_i}^{(k+1)} = w_{y_i}^{(k)} + \phi(x_i)$ \{increase weight of gold class\}
    update $w_{\hat{y}_i}^{(k+1)} = w_{\hat{y}_i}^{(k)} - \phi(x_i)$ \{decrease weight of incorrect class\}
  end if
increment $k$
until maximum number of epochs
output: model weights $w^{(k)}$
Outline

1. Data and Feature Representation

2. Regression

3. Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines

4. Regularization

5. Non-Linear Classifiers
For a moment, forget linear classifiers and parameter vectors $w$.

Let's assume our goal is to model the conditional probability of output labels $y$ given inputs $x$, i.e. $P(y|x)$.

If we can define this distribution, then classification becomes:

$$\hat{y} = \arg \max_{y \in Y} P(y|x)$$
• One way to model $P(y|x)$ is through Bayes Rule:

$$P(y|x) = \frac{P(y)P(x|y)}{P(x)}$$

$$\arg \max_y P(y|x) = \arg \max_y P(y)P(x|y)$$

(since $x$ is fixed!)

• $P(y)P(x|y) = P(x, y)$: a joint probability

• Above is a “generative story”: ‘pick $y$; then pick $x$ given $y$.”

• Models that consider the joint $P(x, y)$ are called generative models, because they come with a generative story.
Naive Bayes

Assume that an input $x$ is partitioned as $v_1, \ldots, v_L$, where $v_k \in \mathcal{V}_k$.

Example:

- $x$ is a document of length $L$
- $v_k$ is the $k^{th}$ token (a word)
- The set $\mathcal{V}_k = \mathcal{V}$ is a fixed vocabulary (all tokens drawn from $\mathcal{V}$)

Naive Bayes Assumption

(conditional independence)

$$P(v_1, \ldots, v_L | y) = \prod_{k=1}^{L} P(v_k | y)$$
Multinomial Naive Bayes

\[ P(x, y) = P(y) P(v_1, \ldots, v_L | y) = P(y) \prod_{k=1}^{L} P(v_k | y) \]

- All tokens are conditionally independently, given the topic
- The word order doesn’t change \( P(x, y) \) (bag-of-words assumption)

**Small caveat:** we assumed that the document has a fixed length \( L \).
This is not realistic.

How to deal with variable length?
**Solution:** introduce a distribution over document length $P(|x|)$

- e.g. a Poisson distribution.

We get:

$$
P(x, y) = P(y) \cdot P(|x|) \cdot \prod_{k=1}^{\frac{|x|}{P(x|y)}} P(v_k|y) \cdot P(x|y)
$$

$P(|x|)$ is constant (independent of $y$), so nothing really changes

- the posterior $P(y|x)$ is the same as before.
What Does This Buy Us?

\[ P(\mathbf{v}_1, \ldots, \mathbf{v}_L | y) = \prod_{k=1}^{L} P(\mathbf{v}_k | y) \]

What do we gain with the Naive Bayes assumption?

• A huge reduction in the number of parameters!
• If we haven't done any factorization assumption, how many parameters would be required for expressing \( P(\mathbf{v}_1, \ldots, \mathbf{v}_L | y) \)?

\( \mathcal{O}(|V| L) \)
• And how many parameters with Naive Bayes?

\( \mathcal{O}(|V|) \)

Less parameters = \( \Rightarrow \) Less computation; less risk of overfitting (Though we may underfit if our independence assumptions are too strong.)
What does this buy us?

$$P(v_1, \ldots, v_L | y) = \prod_{k=1}^{L} P(v_k | y)$$

What do we gain with the Naive Bayes assumption?

- A huge reduction in the number of parameters!
- If we haven’t done any factorization assumption, how many parameters would be required for expressing $P(v_1, \ldots, v_L | y)$?

Less parameters $\Rightarrow$ Less computation; less risk of overfitting (Though we may underfit if our independence assumptions are too strong.)
What Does This Buy Us?

\[ P(v_1, \ldots, v_L | y) = \prod_{k=1}^{L} P(v_k | y) \]

What do we gain with the Naive Bayes assumption?

• A huge reduction in the number of parameters!
• If we haven’t done any factorization assumption, how many parameters would be required for expressing \( P(v_1, \ldots, v_L | y) \)? \( O(|\mathcal{V}|^L) \)
• And how many parameters with Naive Bayes?
What Does This Buy Us?

\[ P(v_1, \ldots, v_L | y) = \prod_{k=1}^{L} P(v_k | y) \]

What do we gain with the Naive Bayes assumption?

- A huge reduction in the number of parameters!
- If we haven’t done any factorization assumption, how many parameters would be required for expressing \( P(v_1, \ldots, v_L | y) \)? \( O(|V|^L) \)
- And how many parameters with Naive Bayes? \( O(|V|) \)
What Does This Buy Us?

\[ P(v_1, \ldots, v_L | y) = \prod_{k=1}^{L} P(v_k | y) \]

What do we gain with the Naive Bayes assumption?

• A huge reduction in the number of parameters!
• If we haven’t done any factorization assumption, how many parameters would be required for expressing \( P(v_1, \ldots, v_L | y) \)? \( O(|\mathcal{V}|^L) \)
• And how many parameters with Naive Bayes? \( O(|\mathcal{V}|) \)

Less parameters \( \Rightarrow \) Less computation; less risk of overfitting

(Though we may underfit if our independence assumptions are too strong.)
Naive Bayes – Learning

\[ P(y)P(v_1, \ldots, v_L | y) = P(y) \prod_{k=1}^{L} P(v_k | y) \]

- Input: dataset \( \mathcal{D} = \{(x_t, y_t)\}_{t=1}^{N} \) (examples assumed i.i.d.)
- Parameters \( \Theta = \{P(y), P(v|y)\} \)
- Objective: Maximum Likelihood Estimation (MLE): choose parameters that maximize the likelihood of observed data

\[
\mathcal{L}(\Theta; \mathcal{D}) = \prod_{t=1}^{N} P(x_t, y_t) = \prod_{t=1}^{N} \left( P(y_t) \prod_{k=1}^{L} P(v_k(x_t)|y_t) \right)
\]

\[
\hat{\Theta} = \arg \max_{\Theta} \prod_{t=1}^{N} \left( P(y_t) \prod_{k=1}^{L} P(v_k(x_t)|y_t) \right)
\]
For the multinomial Naive Bayes model, MLE has a closed form solution!!
It all boils down to counting and normalizing!!
(The proof is left as an exercise...)
\[ \hat{\Theta} = \arg \max_{\Theta} \prod_{t=1}^{N} \left( P(y_t) \prod_{k=1}^{L} P(v_k(x_t) | y_t) \right) \]

\[ \hat{P}(y) = \frac{\sum_{t=1}^{N}[[y_t = y]]}{N} \]

\[ \hat{P}(v | y) = \frac{\sum_{t=1}^{N} \sum_{k=1}^{L}[[v_k(x_t) = v \text{ and } y_t = y]]}{L \sum_{t=1}^{N}[[y_t = y]]} \]

[[X]] is 1 if property X holds, 0 otherwise (Iverson notation)
Fraction of times a feature appears in training cases of a given label
**Naive Bayes Example**

- Corpus of movie reviews: 7 examples for **training**

<table>
<thead>
<tr>
<th>Doc</th>
<th>Words</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Great movie, excellent plot, renown actors</td>
<td>Positive</td>
</tr>
<tr>
<td>2</td>
<td>I had not seen a fantastic plot like this in good 5 years. Amazing!!!</td>
<td>Positive</td>
</tr>
<tr>
<td>3</td>
<td>Lovely plot, amazing cast, somehow I am in love with the bad guy</td>
<td>Positive</td>
</tr>
<tr>
<td>4</td>
<td>Bad movie with great cast, but very poor plot and unimaginative ending</td>
<td>Negative</td>
</tr>
<tr>
<td>5</td>
<td>I hate this film, it has nothing original</td>
<td>Negative</td>
</tr>
<tr>
<td>6</td>
<td>Great movie, but not...</td>
<td>Negative</td>
</tr>
<tr>
<td>7</td>
<td>Very bad movie, I have no words to express how I dislike it</td>
<td>Negative</td>
</tr>
</tbody>
</table>
### Naive Bayes Example

- **Features**: adjectives (bag-of-words)

<table>
<thead>
<tr>
<th>Doc</th>
<th>Words</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Great movie, excellent plot, renowned actors</td>
<td>Positive</td>
</tr>
<tr>
<td>2</td>
<td>I had not seen a fantastic plot like this in good 5 years. amazing !!!</td>
<td>Positive</td>
</tr>
<tr>
<td>3</td>
<td>Lovely plot, amazing cast, somehow I am in love with the bad guy</td>
<td>Positive</td>
</tr>
<tr>
<td>4</td>
<td>Bad movie with great cast, but very poor plot and unimaginative ending</td>
<td>Negative</td>
</tr>
<tr>
<td>5</td>
<td>I hate this film, it has nothing original. Really bad</td>
<td>Negative</td>
</tr>
<tr>
<td>6</td>
<td>Great movie, but not...</td>
<td>Negative</td>
</tr>
<tr>
<td>7</td>
<td>Very bad movie, I have no words to express how I dislike it</td>
<td>Negative</td>
</tr>
</tbody>
</table>
Naive Bayes Example

Relative frequency:

Priors:

\[
P(\text{positive}) = \frac{\sum_{t=1}^{N}[y_t = \text{positive}]}{N} = \frac{3}{7} = 0.43
\]

\[
P(\text{negative}) = \frac{\sum_{t=1}^{N}[y_t = \text{negative}]}{N} = \frac{4}{7} = 0.57
\]

Assume standard pre-processing: tokenization, lowercasing, punctuation removal (except special punctuation like !!!)
Naive Bayes Example

Likelihoods: Count adjective $v$ in class $y$ / adjectives in $y$

$$
\hat{P}(v|y) = \frac{\sum_{t=1}^{N} \sum_{k=1}^{L} [[v_k(x_t) = v \text{ and } y_t = y]]}{L \sum_{t=1}^{N} [[y_t = y]]}
$$

<table>
<thead>
<tr>
<th>Adjective</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazing</td>
<td>$2/10$</td>
<td>$0/8$</td>
</tr>
<tr>
<td>bad</td>
<td>$1/10$</td>
<td>$3/8$</td>
</tr>
<tr>
<td>excellent</td>
<td>$1/10$</td>
<td>$0/8$</td>
</tr>
<tr>
<td>fantastic</td>
<td>$1/10$</td>
<td>$0/8$</td>
</tr>
<tr>
<td>good</td>
<td>$1/10$</td>
<td>$0/8$</td>
</tr>
<tr>
<td>great</td>
<td>$1/10$</td>
<td>$2/8$</td>
</tr>
<tr>
<td>lovely</td>
<td>$1/10$</td>
<td>$0/8$</td>
</tr>
<tr>
<td>original</td>
<td>$0/10$</td>
<td>$1/8$</td>
</tr>
<tr>
<td>poor</td>
<td>$0/10$</td>
<td>$1/8$</td>
</tr>
<tr>
<td>renowned</td>
<td>$1/10$</td>
<td>$0/8$</td>
</tr>
<tr>
<td>unimaginative</td>
<td>$0/10$</td>
<td>$1/8$</td>
</tr>
</tbody>
</table>
Naive Bayes Example

Given a new segment to classify (test time):

<table>
<thead>
<tr>
<th>Doc</th>
<th>Words</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>This was a fantastic story, good, lovely</td>
<td>???</td>
</tr>
</tbody>
</table>

**Final decision**

\[
\hat{y} = \arg \max_y \left( P(y) \prod_{k=1}^{L} P(v_k|y) \right)
\]

\[
P(\text{positive}) \times P(\text{fantastic}|\text{positive}) \times P(\text{good}|\text{positive}) \times P(\text{lovely}|\text{positive})
\]

\[
\frac{3}{7} \times \frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} = 0.00043
\]

\[
P(\text{negative}) \times P(\text{fantastic}|\text{negative}) \times P(\text{good}|\text{negative}) \times P(\text{lovely}|\text{negative})
\]

\[
\frac{4}{7} \times \frac{0}{8} \times \frac{0}{8} \times \frac{0}{8} = 0
\]

So: sentiment = positive
Naive Bayes Example

Given a new segment to classify (test time):

<table>
<thead>
<tr>
<th>Doc</th>
<th>Words</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Great plot, great cast, great everything</td>
<td>???</td>
</tr>
</tbody>
</table>

Final decision

\[
P(\text{positive}) \times P(\text{great}|\text{positive}) \times P(\text{great}|\text{positive}) \times P(\text{great}|\text{positive})
\]

\[
\frac{3}{7} \times \frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} = 0.00043
\]

\[
P(\text{negative}) \times P(\text{great}|\text{negative}) \times P(\text{great}|\text{negative}) \times P(\text{great}|\text{negative})
\]

\[
\frac{4}{7} \times \frac{2}{8} \times \frac{2}{8} \times \frac{2}{8} = 0.00893
\]

So: sentiment = negative
Naive Bayes Example

But if the new segment to classify (test time) is:

<table>
<thead>
<tr>
<th>Doc</th>
<th>Words</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Boring movie, annoying plot, unimaginative ending</td>
<td>???</td>
</tr>
</tbody>
</table>

Final decision

\[
P(\text{positive}) \times P(\text{boring}|\text{positive}) \times P(\text{annoying}|\text{positive}) \times P(\text{unimaginative}|\text{positive})
\]

\[
3/7 \times 0/10 \times 0/10 \times 0/10 = 0
\]

\[
P(\text{negative}) \times P(\text{boring}|\text{negative}) \times P(\text{annoying}|\text{negative}) \times P(\text{unimaginative}|\text{negative})
\]

\[
4/7 \times 0/8 \times 0/8 \times 1/8 = 0
\]

So: sentiment = ???
Laplace Smoothing

Add smoothing to feature counts (add 1 to every count):

$$\hat{P}(v|y) = \frac{\sum_{t=1}^{N} \sum_{k=1}^{L} [v_k(x_t) = v \text{ and } y_t = y] + 1}{L \sum_{t=1}^{N} [y_t = y] + |\mathcal{V}|}$$

where $|\mathcal{V}| = \text{number of distinct adjectives in training (all classes)} = 12$

<table>
<thead>
<tr>
<th>Doc</th>
<th>Words</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Boring movie, annoying plot, unimaginative ending</td>
<td>???</td>
</tr>
</tbody>
</table>

Final decision

$$P(positive) \times P(boring|positive) \times P(annoying|positive) \times P(unimaginative|positive)$$

$$3/7 \times ((0 + 1)/(10 + 12)) \times ((0 + 1)/(10 + 12)) \times ((0 + 1)/(10 + 12)) = 0.000040$$

$$P(negative) \times P(boring|negative) \times P(annoying|negative) \times P(unimaginative|negative)$$

$$4/7 \times ((0 + 1)/(8 + 12)) \times ((0 + 1)/(8 + 12)) \times ((1 + 1)/(8 + 12)) = 0.000143$$

So: sentiment = negative
Finally...

Multinomial Naive Bayes is a Linear Classifier!
Let $b_y = \log P(y)$, $\forall y \in Y$.

Let $[w_y]_v = \log P(v|y)$, $\forall y \in Y, v \in V$.

Let $[\phi(x)]_v = \sum_{k=1}^{L}[[v_k(x) = v]]$, $\forall v \in V$ ($\#$ times $v$ occurs in $x$).

$$\arg\max_y P(y|x) \propto \arg\max_y \left( P(y) \prod_{k=1}^{L} P(v_k(x)|y) \right)$$

$$= \arg\max_y \left( \log P(y) + \sum_{k=1}^{L} \log P(v_k(x)|y) \right)$$

$$= \arg\max_y \left( \underbrace{\log P(y)}_{b_y} + \sum_{v \in V} [\phi(x)]_v \underbrace{\log P(v|y)}_{[w_y]_v} \right)$$

$$= \arg\max_y (w_y \cdot \phi(x) + b_y).$$
• Generative models attempt to model inputs and outputs
  • e.g., Naive Bayes = MLE of joint distribution $P(x, y)$
  • Statistical model must explain generation of input
  • Can we sample a document from the multinomial Naive Bayes model? How?
Discriminative versus Generative

- Generative models attempt to model inputs and outputs
  - e.g., Naive Bayes = MLE of joint distribution \( P(x, y) \)
  - Statistical model must explain generation of input
  - Can we sample a document from the multinomial Naive Bayes model? How?

- Occam’s Razor: why model input?
- Discriminative models
  - Use loss function that directly optimizes \( P(y|x) \) (or something related)
  - Logistic Regression – MLE of \( P(y|x) \)
  - Perceptron and SVMs – minimize classification error
Discriminative versus Generative

• Generative models attempt to model inputs and outputs
  • e.g., Naive Bayes = MLE of joint distribution $P(x, y)$
  • Statistical model must explain generation of input
  • Can we sample a document from the multinomial Naive Bayes model? How?

• Occam’s Razor: why model input?
• Discriminative models
  • Use loss function that directly optimizes $P(y|x)$ (or something related)
  • Logistic Regression – MLE of $P(y|x)$
  • Perceptron and SVMs – minimize classification error

• Generative and discriminative models use $P(y|x)$ for prediction
  • They differ only on what distribution they use to set $w$
Coffee-break!
So far

We have covered:

- The perceptron algorithm
- (Multinomial) Naive Bayes.

We saw that both are instances of linear classifiers.

Perceptron finds a separating hyperplane (if it exists), Naive Bayes is a generative probabilistic model

Next: a discriminative probabilistic model.
\( \hat{y} = \text{argmax} (W \phi(x) + b), \quad W = \begin{bmatrix} \vdots & \vdots \\ w_y^T & \vdots \\ \vdots & \vdots \end{bmatrix}, \quad b = \begin{bmatrix} \vdots \\ b_y \\ \vdots \end{bmatrix} \).
Key Problem

How to map from a set of label scores $\mathbb{R}^{|Y|}$ to a probability distribution over $Y$?

We’ll see two mappings: **softmax** (next) and **sparsemax** (later).
Outline

1 Data and Feature Representation

2 Regression

3 Classification

   Perceptron

   Naive Bayes

   Logistic Regression

   Support Vector Machines

4 Regularization

5 Non-Linear Classifiers
Logistic Regression

Recall: a linear model gives the score for each class, $w_y \cdot \phi(x)$.

Define a conditional probability:

$$P(y|x) = \frac{\exp(w_y \cdot \phi(x))}{Z_x}, \quad \text{where} \quad Z_x = \sum_{y' \in Y} \exp(w_{y'} \cdot \phi(x))$$

This operation (exponentiating and normalizing) is called the softmax transformation (more later!)

Note: still a linear classifier

$$\arg\max_y P(y|x) = \arg\max_y \frac{\exp(w_y \cdot \phi(x))}{Z_x} = \arg\max_y \exp(w_y \cdot \phi(x)) = \arg\max_y w_y \cdot \phi(x)$$
Binary Logistic Regression

Binary labels ($Y = \{ \pm 1 \}$)

Scores: 0 for negative class, $w \cdot \phi(x)$ for positive class

$$P(y = +1 \mid x) = \frac{\exp(w \cdot \phi(x))}{1 + \exp(w \cdot \phi(x))}$$

$$= \frac{1}{1 + \exp(-w \cdot \phi(x))}$$

$$= \sigma(w \cdot \phi(x)).$$

This is called a sigmoid transformation (more later!)
Sigmoid Transformation

\[ \sigma(z) = \frac{1}{1 + e^{-z}} \]

- Widely used in neural networks (wait for tomorrow!)
- Can be regarded as a 2D softmax
- “Squashes” a real number between 0 and 1
- The output can be interpreted as a probability
- Positive, bounded, strictly increasing
Multinomial Logistic Regression

\[ P_W(y \mid x) = \frac{\exp(w_y \cdot \phi(x))}{Z_x} \]

- How do we learn weights \( W \)?
- Set \( W \) to maximize the conditional log-likelihood of training data:

\[
\hat{W} = \arg \max_W \log \left( \prod_{t=1}^{N} P_W(y_t \mid x_t) \right) = \arg \min_W - \sum_{t=1}^{N} \log P_W(y_t \mid x_t) = \\
= \arg \min_W \sum_{t=1}^{N} \left( \log \sum_{y_t'} \exp(w_{y'_t} \cdot \phi(x_t)) - w_{y_t} \cdot \phi(x_t) \right),
\]

i.e., set \( W \) to assign as much probability mass as possible to the correct labels!
Logistic Regression

- This objective function is **convex**
- Therefore any local minimum is a global minimum
- No closed form solution, but lots of numerical techniques
  - Gradient methods (gradient descent, conjugate gradient)
  - Quasi-Newton methods (L-BFGS, ...)

Proof left as an exercise!
• This objective function is convex
• Therefore any local minimum is a global minimum
• No closed form solution, but lots of numerical techniques
  • Gradient methods (gradient descent, conjugate gradient)
  • Quasi-Newton methods (L-BFGS, ...)

• Logistic Regression = Maximum Entropy: maximize entropy subject to constraints on features
• Proof left as an exercise!
Recap: Convex functions

Pro: Guarantee of a global minima ✓

Figure: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.
Recap: Iterative Descent Methods

Goal: find the minimum/minimizer of \( f : \mathbb{R}^d \to \mathbb{R} \)

- Proceed in **small steps** in the **optimal direction** till a **stopping criterion** is met.
- **Gradient descent**: updates of the form: \( x^{(k+1)} \leftarrow x^{(k)} - \eta_k \nabla f(x^{(k)}) \)

**Figure**: Illustration of gradient descent. The red lines correspond to steps taken in the negative gradient direction.
Our loss function in logistic regression is

\[
L(W; (x, y)) = \log \sum_{y'} \exp(w_{y'} \cdot \phi(x)) - w_y \cdot \phi(x).
\]

We want to find \(\operatorname{arg\ min}_W \sum_{t=1}^N L(W; (x_t, y_t))\)

- Set \(W^0 = 0\)
- Iterate until convergence (for suitable stepsize \(\eta_k\)):

\[
W^{k+1} = W^k - \eta_k \nabla_W \left( \sum_{t=1}^N L(W; (x_t, y_t)) \right)
\]

\[
= W^k - \eta_k \sum_{t=1}^N \nabla_W L(W^k; (x_t, y_t))
\]

- \(\nabla_W L(W)\) is gradient of \(L\) w.r.t. \(W\)
- \(L(W)\) convex \(\Rightarrow\) gradient descent will reach the global optimum \(W\).
Stochastic Gradient Descent

It turns out this works with a Monte Carlo approximation of the gradient (more frequent updates, convenient with large datasets):

- Set $W^0 = 0$
- Iterate until convergence
  - Pick $(x_t, y_t)$ randomly
  - Update $W^{k+1} = W^k - \eta_k \nabla W L(W^k; (x_t, y_t))$

- i.e. we approximate the true gradient with a noisy, unbiased, gradient, based on a single sample
- Variants exist in-between (mini-batches)
- All guaranteed to find the optimal $W$ (for suitable step sizes)
Computing the Gradient

- For this to work, we need to compute $\nabla_W L(W; (x_t, y_t))$, where

$$L(W; (x, y)) = \log \sum_{y'} \exp(w_{y'} \cdot \phi(x)) - w_y \cdot \phi(x)$$

- Some reminders:
  1. $\nabla_W \log F(W) = \frac{1}{F(W)} \nabla_W F(W)$
  2. $\nabla_W \exp F(W) = \exp(F(W)) \nabla_W F(W)$

- We denote by

$$e_y = [0, \ldots, 0, \underbrace{1}_{y}, 0, \ldots, 0]^\top$$

the one-hot vector representation of class $y$. 
Computing the Gradient

\[ \nabla_w L(W; (x, y)) = \nabla_w \left( \log \sum_{y'} \exp(w_{y'} \cdot \phi(x)) - w_y \cdot \phi(x) \right) \]

\[ = \nabla_w \log \sum_{y'} \exp(w_{y'} \cdot \phi(x)) - \nabla_w w_y \cdot \phi(x) \]

\[ = \frac{1}{\sum_{y'} \exp(w_{y'} \cdot \phi(x))} \sum_{y'} \nabla_w \exp(w_{y'} \cdot \phi(x)) - e_y \phi(x)^T \]

\[ = \frac{1}{Z_x} \sum_{y'} \exp(w_{y'} \cdot \phi(x)) \nabla_w w_{y'} \cdot \phi(x) - e_y \phi(x)^T \]

\[ = \sum_{y'} \frac{\exp(w_{y'} \cdot \phi(x))}{Z_x} e_{y'} \phi(x)^T - e_y \phi(x)^T \]

\[ = \sum_{y'} P_W(y'|x) e_{y'} \phi(x)^T - e_y \phi(x)^T \]

\[ = \begin{pmatrix} P_W(y'|x) & -e_y \end{pmatrix} \phi(x)^T. \]
Logistic Regression Summary

• Define conditional probability

\[ P_W(y|x) = \frac{\exp(w_y \cdot \phi(x))}{Z_x} \]

• Set weights to maximize conditional log-likelihood of training data:

\[ W = \arg \max_W \sum_t \log P_W(y_t|x_t) = \arg \min_W \sum_t L(W; (x_t, y_t)) \]

• Can find the gradient and run gradient descent (or any gradient-based optimization algorithm)

\[ \nabla_W L(W; (x, y)) = \sum_{y'} P_W(y'|x)e_{y'}\phi(x)^\top - e_y\phi(x)^\top \]
The Story So Far

- Naive Bayes is **generative**: maximizes joint likelihood
  - closed form solution (boils down to **counting and normalizing**)
- Logistic regression is **discriminative**: maximizes conditional likelihood
  - also called log-linear model and max-entropy classifier
  - no closed form solution
  - stochastic gradient updates look like

\[
W^{k+1} = W^k + \eta \left( e_y \phi(x)^\top - \sum_{y'} P_w(y'|x)e_{y'} \phi(x)^\top \right)
\]

- Perceptron is a discriminative, non-probabilistic classifier
  - perceptron’s updates look like

\[
W^{k+1} = W^k + e_y \phi(x)^\top - e_{\hat{y}} \phi(x)^\top
\]

SGD updates for logistic regression and perceptron’s updates look similar!
Maximizing Margin

• For a training set $\mathcal{D}$
• Margin of a weight matrix $W$ is smallest $\gamma$ such that
  \[
  w_{y_t} \cdot \phi(x_t) - w_{y'} \cdot \phi(x_t) \geq \gamma
  \]
• for every training instance $(x_t, y_t) \in \mathcal{D}$, $y' \in \mathcal{Y}$
Denote the value of the margin by $\gamma$
Intuitively maximizing margin makes sense.

More importantly, generalization error to unseen test data is proportional to the inverse of the margin:

$$
\epsilon \propto \frac{R^2}{\gamma^2 \times N}
$$

**Perceptron:**
- If a training set is separable by some margin, the perceptron will find a $W$ that separates the data.
- However, the perceptron does not pick $W$ to maximize the margin!
1. Data and Feature Representation

2. Regression

3. Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines

4. Regularization

5. Non-Linear Classifiers
Let $\gamma > 0$

$$\max_{||U||=1} \gamma$$

such that:

$$u_{y_t} \cdot \phi(x_t) - u_{y'} \cdot \phi(x_t) \geq \gamma$$

$$\forall (x_t, y_t) \in D$$

and $y' \in \mathcal{Y}$

• Note: the solution still ensures a separating hyperplane if there is one (zero training error) – due to the hard constraint
Let $\gamma > 0$

$$\max_{\|U\| = 1} \gamma$$

such that:

$$uy_t \cdot \phi(x_t) - u_{y'} \cdot \phi(x_t) \geq \gamma$$

$$\forall (x_t, y_t) \in D$$

and $y' \in Y$

- **Note:** the solution still ensures a separating hyperplane if there is one (zero training error) – due to the hard constraint
- **We fix** $\|U\| = 1$ since scaling $U$ to increase $\|U\|$ trivially produces larger margin
Let $\gamma > 0$

**Max Margin:**

$$\max_{\|U\|=1} \gamma = \max \quad \gamma$$

such that:

$$u_{y_t} \cdot \phi(x_t) - u_{y'} \cdot \phi(x_t) \geq \gamma$$

$\forall (x_t, y_t) \in D$

and $y' \in \mathcal{Y}$

**Min Norm:**

$$\min_W \frac{1}{2} \|W\|^2$$

such that:

$$w_{y_t} \cdot \phi(x_t) - w_{y'} \cdot \phi(x_t) \geq 1$$

$\forall (x_t, y_t) \in D$

and $y' \in \mathcal{Y}$

- Instead of fixing $\|U\|$ we fix the margin to 1
Let $\gamma > 0$

Max Margin:

$$
\max_{\|U\|=1} \gamma \quad \text{such that:} \quad u_{y_t} \cdot \phi(x_t) - u_{y'} \cdot \phi(x_t) \geq \gamma \\
\forall (x_t, y_t) \in D \\
\text{and } y' \in \mathcal{Y}
$$

Min Norm:

$$
\min_W \frac{1}{2} \|W\|^2 \\
\text{such that:} \quad w_{y_t} \cdot \phi(x_t) - w_{y'} \cdot \phi(x_t) \geq 1 \\
\forall (x_t, y_t) \in D \\
\text{and } y' \in \mathcal{Y}
$$

• Instead of fixing $\|U\|$ we fix the margin to 1
• Make substitution $W = \frac{U}{\gamma}$; then we have $\|W\| = \frac{\|U\|}{\gamma} = \frac{1}{\gamma}$. 
\[ W = \arg \min_W \frac{1}{2} \| W \|^2 \]

such that:

\[ w_{y_t} \cdot \phi(x_t) - w_{y'} \cdot \phi(x_t) \geq 1 \]

\[ \forall (x_t, y_t) \in D \text{ and } y' \in \mathcal{Y} \]

- **Quadratic programming problem** – a well known convex optimization problem
- Can be solved with many techniques.
What if data is not separable?

\[ W = \arg \min_{W, \xi} \frac{1}{2} \| W \|^2 + C \sum_{t=1}^{N} \xi_t \]

such that:

\[ w_{y_t} \cdot \phi(x_t) - w_{y'} \cdot \phi(x_t) \geq 1 - \xi_t \text{ and } \xi_t \geq 0 \]

\[ \forall (x_t, y_t) \in \mathcal{D} \text{ and } y' \in \mathcal{Y} \]

\( \xi_t \): trade-off between margin violations per example and \( \| W \| \)

Larger \( C \) = more examples correctly classified, but smaller margin.
Historically, SVMs with kernels co-occurred together and were extremely popular

Can “kernelize” algorithms to make them non-linear (not only SVMs, but also logistic regression, perceptron, ...)

More later.
Support Vector Machines

\[ W = \arg \min_{W, \xi} \frac{1}{2} \|W\|^2 + C \sum_{t=1}^{N} \xi_t \]

such that:

\[ w_{y_t} \cdot \phi(x_t) - w_{y'} \cdot \phi(x_t) \geq 1 - \xi_t \quad \forall y' \neq y_t \]
\[ W = \arg \min_{W, \xi} \frac{1}{2} \|W\|^2 + C \sum_{t=1}^{N} \xi_t \]

such that:

\[ w_{y_t} \cdot \phi(x_t) - \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) \geq 1 - \xi_t \]
\( \mathbf{W} = \arg \min_{\mathbf{W}, \xi} \left\{ \frac{1}{2} \| \mathbf{W} \|^2 + C \sum_{t=1}^{N} \xi_t \right\} \)

such that:

\[ \xi_t \geq 1 + \max_{y' \neq y_t} \mathbf{w}_{y'} \cdot \phi(x_t) - \mathbf{w}_{y_t} \cdot \phi(x_t) \]
Support Vector Machines

\[ W = \arg \min_{W, \xi} \frac{\lambda}{2} ||W||^2 + \sum_{t=1}^{N} \xi_t \quad \lambda = \frac{1}{C} \]

such that:

\[ \xi_t \geq 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t) \]
\[ W = \arg \min_{W, \xi} \frac{\lambda}{2} \|W\|^2 + \sum_{t=1}^{N} \xi_t \quad \lambda = \frac{1}{C} \]

such that:

\[ \xi_t \geq 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t) \]

If \( W \) classifies \((x_t, y_t)\) with margin 1, penalty \( \xi_t = 0 \)
Otherwise penalty \( \xi_t = 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t) \)
Support Vector Machines

\[
W = \arg \min_{W, \xi} \frac{\lambda}{2} ||W||^2 + \sum_{t=1}^{N} \xi_t \quad \lambda = \frac{1}{C}
\]

such that:

\[
\xi_t \geq 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t)
\]

If \( W \) classifies \((x_t, y_t)\) with margin 1, penalty \( \xi_t = 0 \)
Otherwise penalty \( \xi_t = 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t) \)

Hinge loss:

\[
L((x_t, y_t); W) = \max (0, 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t))
\]
Support Vector Machines

\[ W = \arg \min_{W, \xi} \frac{\lambda}{2} \|W\|^2 + \sum_{t=1}^{N} \xi_t \]

such that:

\[ \xi_t \geq 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t) \]

**Hinge loss equivalent:**

\[ W = \arg \min_W \left( \sum_{t=1}^{N} \max (0, 1 + \max_{y' \neq y_t} w_{y'} \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t)) \right) + \frac{\lambda}{2} \|W\|^2 \]
The hinge loss is a \textit{piecewise linear function}—not differentiable everywhere.

Cannot use gradient descent.

But... can use \textit{subgradient} descent (almost the same)!
Recap: Subgradient

- Defined for convex functions $f : \mathbb{R}^D \rightarrow \mathbb{R}$
- Generalizes the notion of gradient—in points where $f$ is differentiable, there is a single subgradient which equals the gradient
- Other points may have multiple subgradients
Subgradient Descent

\[
L(W; (x, y)) = \max (0, 1 + \max_{y' \neq y} w_{y'} \cdot \phi(x) - w_y \cdot \phi(x)) \\
= \left( \max_{y' \in \mathcal{Y}} w_{y'} \cdot \phi(x) + [y' \neq y] \right) - w_y \cdot \phi(x)
\]

A subgradient of the hinge is

\[
\tilde{\nabla}_W L(W; (x, y)) \ni e_{\hat{y}} \phi(x)^\top - e_y \phi(x)^\top
\]

where

\[
\hat{y} = \arg \max_{y' \in \mathcal{Y}} w_{y'} \cdot \phi(x) + [y' \neq y]
\]

Can also train SVMs with (stochastic) sub-gradient descent!
SVM subgradient update looks like perceptron update

\[ W^{k+1} = W^k - \eta \left\{ \begin{array}{ll}
0, & \text{if } w_y \cdot \phi(x_t) - \max_{y \neq y_t} w_y \cdot \phi(x_t) \geq 1 \\
 e_y \phi(x_t)^\top - e_{y_t} \phi(x_t)^\top, & \text{otherwise, where } y = \arg \max_y w_y \cdot \phi(x_t) + [[y \neq y_t]]
\end{array} \right. \]

Perceptron

\[ W^{k+1} = W^k - \eta \left\{ \begin{array}{ll}
0, & \text{if } w_y \cdot \phi(x_t) - \max_{y \neq y_t} w_y \cdot \phi(x_t) \geq 0 \\
 e_y \phi(x_t)^\top - e_{y_t} \phi(x_t)^\top, & \text{otherwise, where } y = \arg \max_y w_y \cdot \phi(x_t)
\end{array} \right. \]

where \( \eta = 1 \)

Perceptron = SGD with no-margin hinge-loss

\[ \max (0, 1+ \max_{y \neq y_t} w_y \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t)) \]
Summary

What we have covered

• Linear Classifiers
  • Naive Bayes
  • Logistic Regression
  • Perceptron
  • Support Vector Machines

What is next

• Regularization
• Softmax and sparsemax
• Non-linear classifiers
1. Data and Feature Representation

2. Regression

3. Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines

4. Regularization

5. Non-Linear Classifiers
Regularization
Overfitting

If the model is too complex (too many parameters) and the data is scarce, we run the risk of overfitting:

- We saw one example already when talking about add-one smoothing in Naive Bayes!
Regularization

In practice, we regularize models to prevent overfitting

\[
\arg\min_W \sum_{t=1}^{N} L(W; (x_t, y_t)) + \lambda \Omega(W),
\]

where \(\Omega(W)\) is the regularization function, and \(\lambda\) controls how much to regularize.

- Gaussian prior (\(\ell_2\)), promotes smaller weights:

\[
\Omega(W) = \|W\|^2_2 = \sum_y \|w_y\|^2_2 = \sum_y \sum_j w_{y,j}^2.
\]

- Laplacian prior (\(\ell_1\)), promotes sparse weights!

\[
\Omega(W) = \|W\|_1 = \sum_y \|w_y\|_1 = \sum_y \sum_j |w_{y,j}|.
\]
Logistic Regression with $\ell_2$ Regularization

$$
\sum_{t=1}^{N} L(W; (x_t, y_t)) + \lambda \Omega(W) = -\sum_{t=1}^{N} \log \left( \frac{\exp(w_{y_t} \cdot \phi(x_t))/Z_x}{Z_x} \right) + \frac{\lambda}{2} \| W \|^2
$$

- What is the new gradient?

$$
\sum_{t=1}^{N} \nabla_W L(W; (x_t, y_t)) + \nabla_W \lambda \Omega(W)
$$

- We know $\nabla_W L(W; (x_t, y_t))$

- Just need $\nabla_W \frac{\lambda}{2} \| W \|^2 = \lambda W$
Hinge-loss formulation: $\ell_2$ regularization already happening!

$$W = \arg\min_W \sum_{t=1}^{N} L(W; (x_t, y_t)) + \lambda \Omega(W)$$

$$= \arg\min_W \sum_{t=1}^{N} \max (0, 1 + \max_{y \neq y_t} w_y \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t)) + \lambda \Omega(W)$$

$$= \arg\min_W \sum_{t=1}^{N} \max (0, 1 + \max_{y \neq y_t} w_y \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t)) + \frac{\lambda}{2} \|W\|^2$$

↑ SVM optimization ↑
\[ W = \arg\min_W \sum_{t=1}^N L(W; (x_t, y_t)) + \lambda \Omega(W) \]

- **SVMs/hinge-loss:**
  \[ L(W; (x_t, y_t)) = \max(0, 1 + \max_{y \neq y_t} (w_y \cdot \phi(x_t) - w_{y_t} \cdot \phi(x_t))) \]
  \[ \Omega(W) = \frac{1}{2} \|W\|^2 \]

- **Logistic Regression/log-loss:**
  \[ L(W; (x_t, y_t)) = -\log \left( \frac{\exp(w \cdot \psi(x_t, y_t))}{Z_x} \right) \]
  \[ \Omega(W) = \frac{1}{2} \|W\|^2 \]
Loss Function

Should match as much as possible the metric we want to optimize at test time

Should be well-behaved (continuous, maybe smooth) to be amenable to optimization (this rules out the 0/1 loss)

Some examples:

- Squared loss for regression
- Negative log-likelihood (cross-entropy): multinomial logistic regression
- Hinge loss: support vector machines
- Sparsemax loss for multi-class and multi-label classification (next)
Recap

How to map from a set of label scores $\mathbb{R}^{|y|}$ to a probability distribution over $y$?

We already saw one example: softmax.

Next: sparsemax.
Recap: Softmax Transformation

The typical transformation for multi-class classification is softmax:

$$\text{softmax}(\mathbf{z}) = \left[ \frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_{|y|})}{\sum_c \exp(z_c)} \right]$$

- Underlies multinomial logistic regression!
- Strictly positive, sums to 1
- Resulting distribution has full support: $\text{softmax}(\mathbf{z}) > 0, \forall \mathbf{z}$
- A disadvantage if a sparse probability distribution is desired
- Common workaround: threshold and truncate
A sparse-friendly alternative is **sparsemax**: \( \mathbb{R}^{|y|} \rightarrow \Delta^{|y|-1} \), defined as:

\[
\text{sparsemax}(z) := \arg \min_{p \in \Delta^{|y|-1}} \| p - z \|^2.
\]

- In words: Euclidean projection of \( z \) onto the probability simplex
- Likely to hit the boundary of the simplex, in which case \( \text{sparsemax}(z) \) becomes sparse (hence the name)
- Retains many of the properties of softmax (e.g. differentiability), having in addition the ability of producing sparse distributions
- Projecting onto the simplex amounts to a **soft-thresholding** operation
- Efficient linear time forward/backward propagation (see paper)
Sparsemax in Closed Form

- Projecting onto the simplex amounts to a soft-thresholding operation:

\[
\text{sparsemax}_i(z) = \max\{0, z_i - \tau\}
\]

where \(\tau\) is a normalizing constant such that \(\sum_j \max\{0, z_j - \tau\} = 1\)

- To evaluate the sparsemax, all we need is to compute \(\tau\)

- Coordinates above the threshold will be shifted by this amount; the others will be truncated to zero
Two Dimensions

- Parametrize $z = (t, 0)$
- The 2D **softmax** is the logistic (sigmoid) function:
  $$\text{softmax}_1(z) = (1 + \exp(-t))^{-1}$$
- The 2D **sparsemax** is the “hard” version of the sigmoid:
Three Dimensions

- Parameterize $z = (t_1, t_2, 0)$ and plot $\text{softmax}_1(z)$ and $\text{sparsemax}_1(z)$ as a function of $t_1$ and $t_2$
- $\text{sparsemax}$ is piecewise linear, but asymptotically similar to $\text{softmax}$
• Sparsemax is in-between softmax and argmax
• It is \textit{sparse} and \textit{differentiable}.
How to use sparsemax as a loss function?

Caveat: sparsemax is sparse and we don’t want to take the log of zero...
Recap: Multinomial Logistic Regression

- The common choice for a softmax output layer
- The classifier estimates $P(y = c \mid x; W)$
- We minimize the negative log-likelihood:

$$L(W; (x, y)) = -\log P(y \mid x; W) = -\log [\text{softmax}(z(x))]_y,$$

where $z_c(x) = w_c \cdot \phi(x)$ is the score of class $c$.

- Loss gradient:

$$\nabla_W L((x, y); W) = -\left(e_y \phi(x)^T - \text{softmax}(z(x))\phi(x)^T\right)$$
Sparsemax Loss (Martins and Astudillo, 2016)

- The natural choice for a sparsemax output layer
- The neural network estimates $P(y \mid x; W)$ as a sparse distribution
- The sparsemax loss is

$$L((x, y); W) = -z_y(x) + \frac{1}{2} - \frac{1}{2} \| \text{sparsemax}(z(x)) \|^2 + z(x)^\top \text{sparsemax}(z(x)),$$

where $z_y(x) = w_y \cdot \phi(x)$.
- Loss gradient:

$$\nabla_W L((x, y); W) = - \left( e_y \phi(x)^\top - \text{sparsemax}(z(x)) \phi(x)^\top \right)$$
Classification Losses (Binary Case)

- Let the correct label be \( y = +1 \) and define \( s = z_2 - z_1 \).
- Sparsemax loss in 2D becomes a “classification Huber loss”:

![Graph showing various loss functions]
Outline

1. Data and Feature Representation
2. Regression
3. Classification
   - Perceptron
   - Naive Bayes
   - Logistic Regression
   - Support Vector Machines
4. Regularization
5. Non-Linear Classifiers
Recap: What a Linear Classifier Can Do

• It can solve linearly separable problems (OR, AND)
Recap: What a Linear Classifier Can’t Do

... but it can’t solve non-linearly separable problems such as simple XOR (unless input is transformed into a better representation):

This was observed by Minsky and Papert (1969) (for the perceptron) and motivated strong criticisms.
Summary: Linear Classifiers

We’ve seen

- Perceptron
- Naive Bayes
- Logistic regression
- Support vector machines

All lead to convex optimization problems ⇒ no issues with local minima-initialization

All assume the features are well-engineered such that the data is nearly linearly separable
What If Data Are Not Linearly Separable?

Engineer better features (often works!)

Kernel methods:
• works implicitly in a high-dimensional feature space
• ... but still need to choose/design a good kernel
• model capacity confined to positive-definite kernels

Neural networks (next class!)
• embrace non-convexity and local minima
• instead of engineering features/kernels, engineer the model architecture
Engineer better features (often works!)
Engineer better features (often works!)

Kernel methods:
- works implicitly in a high-dimensional feature space
- ... but still need to choose/design a good kernel
- model capacity confined to positive-definite kernels
Engineer better features (often works!)

Kernel methods:
- works implicitly in a high-dimensional feature space
- ... but still need to choose/design a good kernel
- model capacity confined to positive-definite kernels

Neural networks (next class!)
- embrace non-convexity and local minima
- instead of engineering features/kernels, engineer the model architecture
There’s two big ways of building machine learning systems:

1. **Feature-based**: describe objects’ properties (features) and build models that manipulate them
   - everything that we have seen so far.

2. **Similarity-based**: don’t describe objects by their properties; rather, build systems based on *comparing* objects to each other
   - $k$-th nearest neighbors; kernel methods; Gaussian processes.

Sometimes the two are equivalent!
Nearest Neighbor Classifier

- Not a linear classifier!
- In its simplest version, doesn’t require any parameters
- Instead of “training”, **memorize** all the data \( \mathcal{D} = \{(x_i, y_i)_{i=1}^N\} \)
- Given a new input \( x \), find its **most similar** data point \( x_i \) and predict \( \hat{y} = y_i \)
- Many variants (e.g. \( k \)-th nearest neighbor)
- **Disadvantage:** requires searching over the entire training data
- Specialized data structures can be used to speed up search.
A kernel is a similarity function between two points that is **symmetric** and **positive semi-definite**, which we denote by:

\[ \kappa(x_i, x_j) \in \mathbb{R} \]

Given dataset \( \mathcal{D} = \{(x_i, y_i)_{i=1}^N\} \), the **Gram matrix** \( K \) is the \( N \times N \) matrix defined as:

\[ K_{i,j} = \kappa(x_i, x_j) \]

**Symmetric:**

\[ \kappa(x_i, x_j) = \kappa(x_j, x_i) \]

**Positive definite:** for all non-zero \( v \)

\[ vKv^T \geq 0 \]
Kernels

• **Mercer’s Theorem**: for any kernel $\kappa : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$, there exists some feature mapping $\phi : \mathcal{X} \rightarrow \mathbb{R}^\mathcal{X}$, s.t.:

$$\kappa(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

• That is: a kernel corresponds to some a mapping in some **implicit** feature space!

• **Kernel trick**: take a feature-based algorithm (SVMs, perceptron, logistic regression) and replace all explicit feature computations by **kernel evaluations**!

$$w_y \cdot \phi(x) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \alpha_{i,y} \kappa(x, x_i) \quad \text{for some } \alpha_{i,y} \in \mathbb{R}$$

• Extremely popular idea in the 1990-2000s!
Kernels = Tractable Non-Linearity

- A linear classifier in a higher dimensional feature space is a non-linear classifier in the original space.
- Computing a non-linear kernel is sometimes better computationally than calculating the corresponding dot product in the high dimension feature space.
- Many models can be “kernelized” – learning algorithms generally solve the dual optimization problem (also convex).
- Drawback: quadratic dependency on dataset size.
\[ \mathbb{R}^2 \longrightarrow \mathbb{R}^3 \]

\[
(x_1, x_2) \quad \longrightarrow \quad (z_1, z_2, z_3) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)
\]
Popular Kernels

- Polynomial kernel
  \[ \kappa(x_i, x_j) = (\phi(x_i) \cdot \phi(x_j) + 1)^d \]

- Gaussian radial basis kernel
  \[ \kappa(x_i, x_j) = \exp\left(\frac{-\|\phi(x_i) - \phi(x_j)\|^2}{2\sigma}\right) \]

- String kernels (Lodhi et al., 2002; Collins and Duffy, 2002)

- Tree kernels (Collins and Duffy, 2002)
Joint Feature Mappings (useful for the labs)
For multi-class/structured classification, a joint feature map
\( \psi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^D \) is sometimes more convenient

- \( \psi(x, y) \) instead of \( \phi(x) \)

Each feature now represents a joint property of the input \( x \) and the candidate output \( y \).

We’ll use this notation in the labs this afternoon!
Examples

• \( x \) is a document and \( y \) is a label

\[
\psi_j(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains the word "interest"} \\
& \text{and } y = "financial" \\
0 & \text{otherwise}
\end{cases}
\]

\[
\psi_j(x, y) = \% \text{ of words in } x \text{ with punctuation and } y = "scientific"
\]

• \( x \) is a word and \( y \) is a part-of-speech tag

\[
\psi_j(x, y) = \begin{cases} 
1 & \text{if } x = "bank" \text{ and } y = \text{Verb} \\
0 & \text{otherwise}
\end{cases}
\]
More Examples

- $x$ is a name, $y$ is a label classifying the type of entity

\[
\psi_0(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "George"} \\
0 & \text{otherwise}
\end{cases} \quad \psi_4(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "George"} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\psi_1(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "Washington"} \\
0 & \text{otherwise}
\end{cases} \quad \psi_5(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "Washington"} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\psi_2(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "Bridge"} \\
0 & \text{otherwise}
\end{cases} \quad \psi_6(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "Bridge"} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\psi_3(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "General"} \\
0 & \text{otherwise}
\end{cases} \quad \psi_7(x, y) = \begin{cases} 
1 & \text{if } x \text{ contains "General"} \\
0 & \text{otherwise}
\end{cases}
\]

- $x =$ General George Washington, $y =$ Person $\rightarrow \psi(x, y) = [1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0]$
- $x =$ George Washington Bridge, $y =$ Location $\rightarrow \psi(x, y) = [0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0]$
- $x =$ George Washington George, $y =$ Location $\rightarrow \psi(x, y) = [0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0]$
Block Feature Vectors

- $x=$ General George Washington, $y=$ Person $\rightarrow \psi(x, y) = [1\ 1\ 0\ 1\ 0\ 0\ 0\ 0]$
- $x=$ General George Washington, $y=$ Location $\rightarrow \psi(x, y) = [0\ 0\ 0\ 0\ 1\ 1\ 0\ 1]$
- $x=$ George Washington Bridge, $y=$ Location $\rightarrow \psi(x, y) = [0\ 0\ 0\ 0\ 1\ 1\ 1\ 0]$
- $x=$ George Washington George, $y=$ Location $\rightarrow \psi(x, y) = [0\ 0\ 0\ 0\ 1\ 1\ 0\ 0]$

- Each equal size block of the feature vector corresponds to one label
- Non-zero values allowed only in one block
Feature Representations – \( \phi(x) \) vs. \( \psi(x, y) \)

Equivalent if \( \psi(x, y) \) conjoins input features \( \phi(x) \) with one-hot label representations \( e_y := [0, \ldots, 0, 1, 0, \ldots, 0] \)

\[
\psi(x, y) = \phi(x) \otimes e_y \\
= [0, \ldots, 0, \phi(x), 0, \ldots, 0]
\]

\( y \text{-th block} \)

- \( \phi(x) \)
  - \( x=\text{General George Washington} \rightarrow \phi(x) = [1 \ 1 \ 0 \ 1] \)

- \( \psi(x, y) \)
  - \( x=\text{General George Washington, } y=\text{Person} \rightarrow \psi(x, y) = [1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0] \)
  - \( x=\text{General George Washington, } y=\text{Object} \rightarrow \psi(x, y) = [0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1] \)

\( \phi(x) \) is sometimes simpler and more convenient in binary classification

... but \( \psi(x, y) \) is more expressive (allows more complex features over properties of labels)
Linear Classifiers – $\psi(x, y)$

- Parametrized by a weight vector $w \in \mathbb{R}^D$ (one weight per feature)
- The score (or probability) of a particular label is based on a linear combination of features and their weights
- At test time (known $w$), predict the class $\hat{y}$ which maximizes this score:
  $$\hat{y} = h(x) = \arg \max_{y \in Y} w \cdot \psi(x, y)$$
- At training time, different strategies to learn $w$ yield different linear classifiers: perceptron, naïve Bayes, logistic regression, SVMs, ...
• Define $|\mathcal{Y}|$ weight vectors $\mathbf{w}_y \in \mathbb{R}^D$
  • i.e., one weight vector per output label $y$

• Classification

$$\hat{y} = \arg \max_{y \in \mathcal{Y}} \mathbf{w}_y \cdot \phi(x)$$
Linear Classifiers – $\phi(x)$

• Define $|\mathcal{Y}|$ weight vectors $\mathbf{w}_y \in \mathbb{R}^D$
  • i.e., one weight vector per output label $y$

• Classification

$$\hat{y} = \arg \max_{y \in \mathcal{Y}} \mathbf{w}_y \cdot \phi(x)$$

• $\psi(x, y)$
  • $x=$ General George Washington, $y=$ Person $\rightarrow \psi(x, y) = [1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$
  • $x=$ General George Washington, $y=$ Object $\rightarrow \psi(x, y) = [0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1]$
  • Single $\mathbf{w} \in \mathbb{R}^8$

• $\phi(x)$
  • $x=$ General George Washington $\rightarrow \phi(x) = [1 \ 1 \ 0 \ 1]$
  • Two parameter vectors $\mathbf{w}_0 \in \mathbb{R}^4$, $\mathbf{w}_1 \in \mathbb{R}^4$
Conclusions

- Linear classifiers are a broad class including well-known ML methods such as perceptron, Naive Bayes, logistic regression, support vector machines
- They all involve manipulating weights and features
- They either lead to closed-form solutions or convex optimization problems (no local minima)
- Stochastic gradient descent algorithms are useful if training datasets are large
- However, they require manual specification of feature representations
- **Tomorrow:** methods that are able to learn internal representations
Thank You!

Questions?


