Linear Models

Linear models are a class of **regression** and **classification** models in which the prediction is a linear function of the input variables.

- Also known as **linear classifiers**;
- A component of **neural networks**.

Summary:

Feature Representations

- **Feature representation** is the process of transforming raw data into a set of features that can be used to more effectively train a machine learning model;
- Bag-of-words features - used for text classification;
- SIFT features - used for image classification;
- Other categorical, boolean and continuous features.

To represent information about x, a typical approach is to define a **feature map** $\phi : X \to \mathbb{R}^d$ that maps x to a d-dimensional feature vector $\phi(x)$ - **feature vector**;

- $\phi(x)$ is a vector of d features;
- It may include boolean, categorical and continuous features;
- Categorical features can be encoded as **one-hot vectors** - a vector with a 1 in the position corresponding to the category and 0s everywhere else.

NLP (Natural Language Processing) Pipelines

- Classical NLP pipelines: stacking together several linear classifiers;
- Each output is used to **handcraft features** for the next classifier - **feature engineering**.

Linear Regression

- Output space Y is **continuous** (e.g. $\mathbb{R}$);
- Model: $\hat{y} = w^T x + b$;
- w is a d-dimensional vector of **weights**;
- b is a **bias**;
- Given **training data** $D = \{(x_n, y_n)\}_{n=1}^N$, we want to find the **best** w and b.

The **least squares** approach is to find w and b that minimize the **mean squared error** (MSE):

$$min_{w,b} \sum_{n=1}^{N} (y_n - \hat{y}_n)^2 = min_{w,b} \sum_{n=1}^{N} (y_n - (w^T x_n + b))^2$$

- **Maximum likelihood estimate** (MLE) of w and b.

Closed-Form Solution

- Ofter, linear dependency of $\hat{y}$ on x is a bad assumption, so a more general approach is to usa a **feature vector** $\phi(x)$:

$$\hat{y} = w^T \phi(x)$$

- The **bias** term is included in $\phi(x)$, in thee constant feature $\phi_0(x) = 1$;
- **Fit/learn** w by solving:

$$min_w \sum_{n=1}^{N} (y_n - w^T \phi(x_n))^2$$

- The **closed-form solution** is:

$$\hat{w} = (X^T X)^{-1} X^T y, with X = \begin{bmatrix} \phi(x_1)^T \\ \vdots \\ \phi(x_N)^T \end{bmatrix}, y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

Squared Loss Function

The **squared loss function** is:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2, with \hat{y} = w^T \phi(x)$$

The model is fit to training data by **minimizing the loss function**:

$$\hat{w} = argmin_w \sum_{n=1}^{N} L(y_n - \hat{y}_n) = argmin_w \sum_{n=1}^{N} \frac{1}{2}(y_n - w^T \phi(x_n))^2$$

The $\frac{1}{2}$ factor is included for **convenience**, so that the derivative of the loss function is $y - \hat{y}$.

Other Regression Loss Functions

- **Squared loss** (most common): $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$;
- **Absolute error loss**: $L(y, \hat{y}) = |y - \hat{y}|$;
- **Huber loss**: $L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$.

Overfitting and Underfitting

- **Overfitting**: the model is **too complex** for the data;
  - To avoid overfitting:
    - **Regularization** - penalize large weights;
    - Some way to choose the number of features D;
- **Underfitting**: the model is **too simple** for the data.

Ridge Regression and Regularization

If X^TX is **not invertible**, we can add a **regularization term** to the loss function, using the **ridge regression** approach:

$$\hat{w}_{ridge} = (X^TX + \lambda I)^{-1}X^Ty$$

This is equivalent to (considering $||w||_2^2 = \sum_{i=1}^{d} w_i^2$, the **L2 norm**):

$$\hat{w}_{ridge} = argmin_w||Xw - y||^2 + \lambda||w||_2^2$$

- l_2 regularization is also known as **weight decay**, or penalized least squares.

Maximum A Posteriori (MAP)

- **Maximum a posteriori** (MAP) is a **Bayesian** approach to learning the parameters of a model;
- Assuming that we have a **prior distribution** $w \sim \mathcal{N}(0, \tau^2 I)$, we can find the **maximum a posteriori** estimate of w:

$$\hat{w}_{MAP} = argmax_w p(w|y) = argmax_w p(y|w)p(w)/p(y) = argmin_w \lambda||w||_2^2 + \sum_{n=1}^{N}(y_n - w^T\phi(x_n))^2$$

- The **prior** $||w||_2^2$ is the **regularization term** - regularizer;
- The **likelihood** $\sum_{n=1}^{N}(y_n - w^T\phi(x_n))^2$ is the **loss function**;
- The **regularization constant** is $\lambda = \frac{\sigma^2}{\tau^2}$.

Binary Classification

- Output space Y is **binary** (e.g. $\{0, 1\}$);
- Model:

$$\hat{y} = sign(w^T\phi(x) + b) = \begin{cases} +1 & \text{if } w^T\phi(x) + b \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

- **Sign function** converts from continuous to binary; this is different from regression;
- The decision boundary is a **hyperplane**: $w^T\phi(x) + b = 0$;
- The dataset D is **linearly separable** if there exists a pair (w, b) such that classification is **perfect**.

Perceptron Learning Algorithm

- **Perceptron learning algorithm** (PLA) is an **iterative** algorithm for finding a **linear classifier**;
- Process one data point x_n at each round:
  1. Apply the current model to x_n to get a prediction $\hat{y}_n$;
  2. If $\hat{y}_n$ is correct, do nothing;
  3. If $\hat{y}_n$ is incorrect, update the model, by **adding/subtracting** feature vector $\phi(x_n)$ to/from w;
- The training data is **linearly separable** with **margin** $\gamma > 0$ if there exists a weight vector u, with $||u|| = 1$, such that $y_n(u^T\phi(x_n)) \geq \gamma$ for all n;
- **Radius** of the data is $R = max_n||\phi(x_n)||$;
- The following bound of **number of mistakes** k is valid: $k \leq (\frac{R}{\gamma})^2$.

But what a perceptron can and cannot do?

- Since is a **linear classifier**, it cannot solve **non-linearly separable** problems, such as XOR;
- But it can solve **linearly separable** problems, such as AND, OR and NAND.

Multiclass Classification

- Assume now a **multi-class** problem, with $|Y| > 2$ labels (classes);
- There are several strategies to **reduce to binary classification**:
  - **One-vs-all** (OVA): train $|Y|$ classifiers, each one to distinguish between one class and the rest;
  - **One-vs-one** (OVO): train $\frac{|Y|(|Y|-1)}{2}$ classifiers, each one to distinguish between one pair of classes;
  - **Binary coding**: use a binary code (maybe a **error-correcting code**) to represent each class, and train a binary classifier for each bit.

Multiclass Linear Classifiers

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

$$W = \begin{bmatrix} w_1 & \cdots & w_{|Y|} \end{bmatrix}, b = \begin{bmatrix} b_1 & \cdots & b_{|Y|} \end{bmatrix}$$

- The score of a particular label is based on a **linear combination** of the features and the corresponding weights;
- Predict the label $\hat{y}$ with the **highest score**:

$$\hat{y} = argmax_{y \in Y}(W\phi(x) + b)$$

Logistic Regression

- A linear model gives a **score** for each class y: $w_y^T \phi(x)$, from which we may compute a **conditional posterior probability** $p(y|x) = \frac{exp(w_y^T \phi(x))}{\sum_{y' \in Y} exp(w_{y'}^T \phi(x))}$;
  - This is known as **softmax transformation**;
  - We choose the **most probable class**: $argmax_{y \in Y} p(y|x) = argmax_{y \in Y}(w_y^T \phi(x))$;
- **Sigmoid transformation**:

$$\sigma(z) = \frac{e^u}{1 + e^u}$$

- Widely used in neural networks;
- Maps $\mathbb{R}$ to $[0, 1]$;
- The output can be interpreted as a **probability**;
- Positive, bounded and strictly increasing function.

Multinomial Logistic Regression

- **Multinomial logistic regression** is a **generalization** of logistic regression to **multiple classes**;

$$P_W(y|x) = \frac{exp(w_y^T \phi(x))}{\sum_{y' \in Y} exp(w_{y'}^T \phi(x))}$$

- Maximize the **conditional log-likelihood** of the training data, in order to find the best weights W:

$$\hat{W} = argmax_W \sum_{n=1}^{N} logP_W(y_n|x_n)$$

- Function **strictly convex** - any local minimum is also a global minimum;
- No closed-form solution, but can be solved with **gradient descent**, or other optimization methods.

Gradient Descent Recap

- Goal: **minimize** a function $f : \mathbb{R}^d \to \mathbb{R}$, for differentiable f;
- Take **small steps** in the **negative gradient direction**, until **stopping criterion** is met: $w^{(t+1)} = w^{(t)} - \eta \nabla f(w^{(t)})$.

The **loss function** in logistic regression is:

$$L(W; (x, y)) = log \sum_{y' \in Y} exp(w_{y'}^T \phi(x)) - w_y^T \phi(x)$$

Stochastic Gradient Descent

- **Stochastic gradient descent** (SGD) is a variant of gradient descent, where the **gradient** is **approximated** using a **single data point**;
- **Monte Carlo approximation** - more frequent updates, which is convenient with large datasets;

1. Set $W^{(0)} = 0$;
2. Iterate until stopping criterion is met:
   (a) Pick a random data point (x_n, y_n);
   (b) Update $W^{(t+1)} = W^{(t)} - \eta \nabla L(W^{(t)}; (x_n, y_n))$.

- Approximate the gradient with a **single data point**;
- **Noisy** updates, but **faster**;
- **Mini-batch SGD**: use a **small batch** of data points to approximate the gradient.

Regularization

- **Regularization** is a technique to **reduce overfitting** - when the model is too complex for the data;

$$\hat{w} = argmin_w \sum_{n=1}^{N} L(y_n - w^T \phi(x_n)) + \lambda \Omega(w)$$

- $\Omega(w)$ is the **regularization term**, and λ is the **regularization constant** that controls the weight of the regularization term;
- l_2 regularization: $\Omega(w) = ||w||_2^2 = \sum_{i=1}^{d} w_i^2$ - promotes **smaller weights**;
- l_1 regularization: $\Omega(w) = ||w||_1 = \sum_{i=1}^{d} |w_i|$ - promotes **smaller and sparse weights**.

Non-Linear Models

- **Linear models** are **limited** to linear decision boundaries - data must be **linearly separable**;
- **Non-linear models** can learn **non-linear decision boundaries**;
- There are several approaches to **non-linear models**:
  - **Feature engineering** - manually define non-linear features;
  - **Kernel methods** - implicitly map data to a higher-dimensional space;
  - **Neural networks** - learn non-linear features.

There are two main ways of building machine learning systems:

1. **Feature-based** - describe object properties (features) and build a model that uses these features - **feature engineering**;
2. **Similarity-based** - define a similarity measure between objects and build a model that uses this similarity - **kernel methods**, **knn**.

KNN (K-Nearest Neighbors) Classifier

- **KNN** is a **non-parametric** classifier;
- No training is required, just **memorize** the training data;
- Given a new input x, find the k **closest** training examples x_n and **predict** the **majority label** among them: $\hat{y(x)} = y_n, wheren = argmin||x - x_i||$;
- **Disadvantage**: **expensive** at test time - must compute the distance to all training examples.

Kernel Methods

- A kernel is a **similarity function** $k : X \times X \to \mathbb{R}$, that is **symmetric and positive semi-definite**;
- Given dataset $D = \{(x_n, y_n)\}_{n=1}^{N}$, we can define the **Gram matrix** $K \in \mathbb{R}^{N \times N}$, where $K_{ij} = k(x_i, x_j)$;
  - **Symmetric** - $K_{ij} = K_{ji}$;
  - **Positive semi-definite** - $v^T K v \geq 0$ for all $v \in \mathbb{R}^N$;
- **Mercer's theorem**: a function k is a valid kernel if and only if K is a valid Gram matrix;
- **Kernel trick**: take a feature-based algorithm (e.g. linear regression) and replace the dot product $x^T x'$ with a kernel $k(x, x')$. The resulting algorithm is a **non-linear** algorithm that operates in the **feature space** - many models can be **kernelized**.

There are several popular kernels:

- **Polynomial kernel**: $k(x, x') = (x^T x' + c)^d$;
- **Gaussian radial basis function (RBF)**: $k(x, x') = exp(-\frac{||x-x'||^2}{2\sigma^2})$;
- **String kernels**;
- **Tree kernels**.