Lecture 5

Statistical Methods for Chemical Prediction

Contents

In this second lecture on statistical analysis, we introduce a set of numerical methods for prediction. These methods can be used to predict chemical, physical, and biological properties, as well as experimental outcomes and related quantities. Such predictive models enable high-throughput virtual screening (HTVS), improve the accuracy of computational approaches, and help guide experimental design.

Although the methods introduced in this lecture are precursors of deep learning methods, you can already perform research-level data analysis and chemical predictions with methods introduced in this lecture. For example, regression methods has been widely used in bioproperty predictions, and Bayesian methods are widely used for experimental design. In many cases, we do not have access to big data, and therefore, using complicated large models will introduce overfitting problems. The methods introduced in this lecture, and their improved versions (for curious readers), are suitable for small datasets.

1 Linear Models

In this section, we focus on linear models for both regression and classification. By definition

- **Regression**: predicting a *continuous* numerical output: $y \in \mathbb{R}$.
- **Classification**: predicting a discrete/categorical label: $y \in \{0, 1, \cdots, C-1\}$, where C is the number of categories.

Examples in Chemistry:

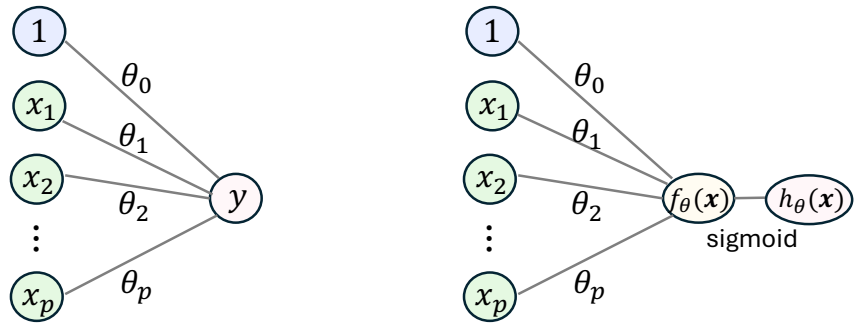
- Regression: solubility, melting point, reaction yield, $\log P$, band gap, etc.
- Classification: toxic vs. non-toxic, reactive vs. inert, and hydrophilic vs. hydrophobic compounds, solid/liquid/gas phases, drug-like vs. non-drug-like, etc.

Continuous properties can also be converted into categorical labels. For example, solubility values may be discretized into low, medium, and high classes by defining appropriate thresholds.

1.1 Learning Objectives

We aim to model the relationship between a set of input variables $\mathbf{x}$ and an output variable y, which depends on the values of $\mathbf{x}$. The input $\mathbf{x}$ may be a vector consisting of multiple quantities, while the output y is typically a single scalar value (continuous or discrete).

The goal is to construct a parametrized function f_θ such that the prediction $f_\theta(\mathbf{x})$ is close to the observed value y.



(a) Linear Regression (b) Logistic Regression

Figure 1: Neural-network form of (a) linear regression and (2) logistic regression.

In practice, we work with many observed input-output pairs,

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\},$$

and seek a function f_θ whose predictions are close to the corresponding targets across the dataset.

In statistical terminology, y is called the *dependent variable* (also referred to as the response or outcome), while the components of $\mathbf{x}$ are called *independent variables* (often referred to as predictors, covariates, or features). This terminology reflects the modeling assumption that y depends on $\mathbf{x}$.

In this section, we introduce two methods: **linear regression** for continuous value prediction, and **logistic regression** for classification. Structures of the two methods are shown in Fig. 1.

1.2 Linear Regression

The linear regression model is the simplest and widely used regression model. It has a very simple form

$$y \approx f_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \cdots \theta_p x_p, \tag{1}$$

where p is the length of the independent variable $\mathbf{x}$. The linear regression model provides the simplest neural network form: a one layer neural network, shown in Fig. 1 (a).

Now let's define what "close" means here. For n pairs of $(\mathbf{x}, y)$, we determine the value of θ such that

$$\arg\min_\theta \sum_{i=1}^{n} \left(y_i - f_\theta(\mathbf{x}_i)\right)^2, \tag{2}$$

which is called *least squares method*. We use the term **loss function** to represent the function to minimize, and here

$$\text{Loss} = \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2 \tag{3}$$

Can we connect Eq. (2) **to maximum likelihood estimation (MLE)?**

Yes. Let's first turn Eq. (2) into a distribution, say

$$\begin{aligned} p(\{\mathbf{x}_i, y_i\}|\theta) &= \frac{1}{\lambda} \prod_{i=1}^{n} \exp\left[-\frac{(y_i - f_\theta(\mathbf{x}_i))^2}{\eta^2}\right] \\ &= \frac{1}{\lambda} \prod_{i=1}^{n} \exp\left[-\frac{\left[(y_i - \theta_1 x_{i,1} - \cdots \theta_p x_{i,p}) - \theta_0\right]^2}{\eta^2}\right], \end{aligned}$$

where λ is the normalization factor, and η is used to adjust the width of the distribution. $p(\{\mathbf{x}_i, y_i\}|\theta)$ is a *joint distribution* with $(p + 1)$ random variables, with parameters θ, and it looks like a Gaussian distribution. Then $p(\{\mathbf{x}_i, y_i\}|\theta)$ has the form of a likelihood $L(\theta|\{\mathbf{x}_i, y_i\})$. From last lecture, we learned that maximizing $L(\theta|\{\mathbf{x}_i, y_i\})$ is equivalent to minimizing $\sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2$.

Why the name *Regression?* Regression is called "regression" because it originally described the tendency of offspring's heights to revert toward the population mean: "regression toward the mean", by Francis Galton.

1.2.1 QSAR

QSAR (Quantitative Structure–Activity Relationship) modeling is applying regression analysis to learn the relationship between molecular structures and biological or physicochemical property, with the following workflow

$$\text{molecular structure} \xrightarrow{\text{descriptors}} \mathbf{x} \xrightarrow{\text{regression}} y$$

The target y to learn are continuous-valued properties, including

- Physicochemical properties: boiling point, solubility, density, viscosity
- Thermodynamic quantities: free energies, enthalpies
- biological activities: binding affinity, $\log(\text{IC}_{50})$[1], inhibition constants
- environmental properties: toxicity, bioaccumulation, degradation rates

The molecular descriptors $\mathbf{x}$ are those easier to get given the molecular structure, e.g.

[1] log-transformed measure of biological activity

- Constitutional descriptors: molecular weight, number of atoms, number of rings
- Topological descriptors: connectivity indices, graph-based measures
- Geometric descriptors: molecular volume, surface area
- Electronic descriptors: partial charges, dipole moments
- Fingerprints: high-dimensional binary or count-based vectors encoding substructures

Regularization is essential for QSAR, since many descriptors are needed, and these descriptors are usually correlated, while the number of samples n is limited due to experimental cost.

While QSAR is powerful, predictions are only reliable within the domain of the training data, and the descriptor choice strongly influences model performance. Linear models cannot capture complex nonlinear structure–property relationships These limitations motivate more advanced methods, such as nonlinear regression, classification models, and probabilistic approaches.

In Lab 5, we will see an example of applying regression to predict the solubility.

1.3 Logistic Regression

Logistic regression shares the linear backbone of linear regression, but adds a non-linear transformation to map a real-valued score to a binary outcome (0 or 1). Its structure is illustrated in Fig. 1 (b).

We first define a linear score

$$f_\theta(\mathbf{x}) = \theta_0 + \sum_{j=1}^{p} \theta_j x_j. \tag{4}$$

The probability of the positive class is then given by a *sigmoid* (logistic) function:

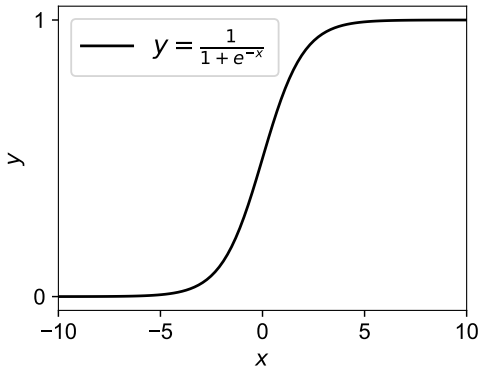


Figure 2: Sigmoid function.

$$P(y = 1 \mid \mathbf{x}) = h_\theta(\mathbf{x}) = \frac{1}{1 + \exp(-f_\theta(\mathbf{x}))}, \tag{5}$$

and the probability of the negative class is

$$P(y = 0 \mid \mathbf{x}) = 1 - h_\theta(\mathbf{x}). \tag{6}$$

The sigmoid function $\sigma(z) = 1/(1 + e^{-z})$ acts as a smooth approximation to a step function, as shown in Fig. 2. Notably, the Fermi-Dirac distribution in statistical mechanics has the same functional form.

To convert predicted probabilities into class labels, a threshold is applied. The most common choice is

$$\begin{cases} 1, & \text{if } h_\theta(\mathbf{x}) \geq 0.5, \\ 0, & \text{if } h_\theta(\mathbf{x}) < 0.5. \end{cases} \tag{7}$$

The threshold can be adjusted to trade off false positives and false negatives, depending on the application.

1.3.1 Loss function

Although one could apply a least-squares loss, it is not well suited for probabilistic classification. Instead, logistic regression is trained by minimizing the **cross-entropy loss**, which corresponds to the negative log-likelihood of a Bernoulli model:

$$\text{Loss}(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \ln h_\theta(\mathbf{x}_i) + (1 - y_i) \ln(1 - h_\theta(\mathbf{x}_i)) \right]. \tag{8}$$

Here, $\{y_i\}$ are the true labels and $\{h_\theta(\mathbf{x}_i)\}$ are the predicted probabilities.

In practice, numerical issues may arise when $h_\theta(\mathbf{x}_i)$ becomes extremely close to 0 or 1. To ensure numerical stability, predictions are commonly clipped using a small threshold $0 < \epsilon \ll 1$:

- if $h_\theta(\mathbf{x}_i) < \epsilon$, set $h_\theta(\mathbf{x}_i) = \epsilon$;
- if $h_\theta(\mathbf{x}_i) > 1 - \epsilon$, set $h_\theta(\mathbf{x}_i) = 1 - \epsilon$.

1.4 Performance Metrics

In machine learning, a **score** is a metric used to evaluate the performance of a model. Different types of models have different commonly used scores. Here are the main ones for the methods we have covered:

- **Linear regression:** R^2 score (↑), RMSE (↓), and MAE (↓)
- **Logistic regression:** Accuracy (↑), Precision (↑), Recall (↑), F1 score (↑), and ROC AUC (↑)

These scores are readily available in standard machine learning packages, but it is useful to understand what they measure. For detailed definitions and formulas, see Appendix B.

1.5 Overfitting

Overfitting happens when more parameters than need are used in a prediction function, which introduces redundancy. When does overfitting happen for linear regression or logistic regression?

Let's look at an example, in the predicting boiling point example, we had two independent variables, molecular weight (x_1) and viscosity (x_2). Now let's add another feature, the density of the liquid (x_3). We know that the density should be roughly proportional to molecular weight, so they are largely correlated. Let's assume $x_3 = ax_1$, then the linear regression model becomes

$$f_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 a x_1 = \theta_0 + (\theta_1 + a\theta_3)x_1 + \theta_2 x_2,$$

which means we have 3 knowns and 4 unknowns! Then it is hard to determine the θ value. What's worse? If $a > 0$, we can have a very large θ_1 and very negative θ_3, and their sum $(\theta_1 + a\theta_3)$ can still be some finite number!

Therefore, overfitting for linear regression happens when there are dependencies among the independent variables. To overcome this problem, we could remove the dependency (correlation), which will be discussed in the next lecture. We could also use the **regularization** technique to prevent overfitting.

In addition, when the scales of the feature values differ too much, e.g., x_1 being on the order of 10^3 while x_2 is on the order of 10^{-3}, the fitting can also be problematic, because the parameters can lose their numerical stability and interpretability. In practice, features with larger scales tend to dominate the optimization, leading to ill-conditioned normal equations and slow or unstable convergence in gradient-based methods. Rescaling is needed for the training stability.

In summary, three important considerations when applying linear regression are:

- **Dimension reduction:** reduce the number of features to avoid overfitting and improve interpretability (covered in the next lecture).
- **Rescaling:** standardize or normalize features so that all variables contribute comparably to the model.
- **Regularization:** apply L1 or L2 penalties to prevent overfitting when the number of features is large or features are correlated.

1.5.1 Regularization

Regularization prevents overfitting by adding penalty terms to the least square cost function Eq. (2), to penalize large coefficients. We will see two regularization strategies: L1 and L2 regularization, leading to Lasso regression and Ridge regression, respectively.

- L1 regularization ⇒ Lasso regression.
- L2 regularization ⇒ Ridge regression.

We simply add the 1st order and 2nd order penalty terms, to the least square loss

$$\begin{aligned} &\text{Lasso (L1):} \quad \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2 + \lambda \sum_{j=1}^{p} |\theta_j| \\ &\text{Ridge (L2):} \quad \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2 + \lambda \sum_{j=1}^{p} \theta_j^2, \end{aligned} \tag{9}$$

where $\lambda > 0$ is the coefficient that controls the penalty strength.

1.6 Practical Considerations in Training

When applying machine learning methods mentioned above, several practical considerations are essential to achieve reliable and generalizable models.

1. **Train-Test Splitting.** A fundamental step in supervised learning is splitting the data into training and testing sets. The training set is used to fit the model, while the test set evaluates its generalization performance. A common split is 70%–80% for training and 20%–30% for testing.
2. **Validation.** Similar idea to train-test splitting. For small datasets, **cross-validation** is recommended. In k-fold cross-validation, the data is split into k subsets; each subset is used once as the validation set while the remaining $k - 1$ subsets are used for training. The performance is averaged over all folds to provide a more robust estimate of model generalization.
3. **Bias-Variance Tradeoff.** Understanding the bias-variance tradeoff is critical in model selection:

    - **High bias** models (e.g., linear regression with too few features) may underfit the data, failing to capture important patterns.
    - **High variance** models (e.g., deep trees in random forests or high-degree polynomial regression) may overfit the training data, capturing noise instead of the underlying signal.

    Regularization techniques, such as L1/L2 penalties in regression or pruning in decision trees, help control variance while maintaining predictive power.
4. **Hyperparameter Tuning.** Most models have hyperparameters that significantly affect performance, e.g., regularization strength in logistic regression, number of neighbors in kNN, or kernel parameters in SVM and Gaussian processes. As the course unrolls, we will

run into many hyperparameters, such as the learning rate, and damping terms.

5. **Data Leakage** Ensure that no information from the test set leaks into training. This includes scaling, feature selection, or hyperparameter tuning; all transformations must be fitted only on the training data and applied to the test data.

2 Bayesian Methods

Bayesian methods are particularly effective in small-data regimes, such as experimental design, because they incorporate prior knowledge and naturally regularize models. Unlike conventional regression approaches, Bayesian inference provides a principled framework for quantifying uncertainty in predictions. Common applications include chemical property prediction, hyperparameter optimization, and experimental design.

Bayesian methods can also be viewed as a conceptual precursor to *generative models*, as they define a joint probability model and perform inference over latent variables.

What sets Bayesian methods apart in statistical learning is that they do not rely on optimizing explicit cost functions. Instead, they encode assumptions through prior distributions, and predictions arise naturally from Bayes' theorem and these assumptions.

In this lecture, we focus on understanding Bayesian inference and a widely used method in chemistry, the **Gaussian process**. In subsequent lectures, we will explore Bayesian optimization, a technique for efficiently optimizing expensive or difficult-to-evaluate functions—such as chemical experiments, reaction conditions, or material properties—by combining probabilistic modeling with sequential decision-making.

2.1 Bayes' Theorem

Bayesian inference is based on Bayes' theorem for conditional probability:

$$\text{Bayes' Theorem:} \quad P(Y|X) = \frac{P(X|Y)\,P(Y)}{P(X)}. \tag{10}$$

This relationship follows directly from the joint probability chain rule:

$$\text{Joint Probability:} \quad P(X,Y) = P(X|Y)\,P(Y) = P(Y|X)\,P(X). \tag{11}$$

In practice, if Y is the target variable and X is observed data, Bayesian methods allow us to compute the conditional distribution of Y given X. Instead of learning a single deterministic mapping from X to Y, we model *the full uncertainty* over Y conditioned on X.

Now let's properly introduce Bayesian methods for parameter inference and posterior computation.

2.2 Bayesian Inference

Since Bayesian inference more general than simple prediction tasks, we use abstract symbols:

- D : the observed data.
- θ : the parameters we want to learn. These can represent model parameters, experimental hyperparameters, or target values in prediction tasks.

Our goal is to compute the conditional distribution $p(\theta|D)$ of the parameters given the data:

$$p(\theta|D) = \frac{p(D|\theta)\, p(\theta)}{p(D)}. \tag{12}$$

In Eq. (12), we need to specify two components based on our modeling of the problem, $p(\theta)$ and $p(D|\theta)$:

- $p(\theta)$, called the prior, encodes our belief about θ before seeing any data. For example, we might choose a wide uniform distribution if we have little prior knowledge. This distribution is used at the very beginning. After observing data D, it is updated to $p(\theta|D)$, which can then serve as the prior in the next updating step if new data arrive.
- $p(D|\theta)$ describes the probability of observing the data assuming a given parameter value θ. This comes from our model of how the data are generated.

Once $p(\theta)$ and $p(D|\theta)$ are specified, $p(\theta|D)$ is obtained by combining them using Bayes' theorem. The denominator $p(D)$ is a normalization factor that ensures $p(\theta|D)$ integrates to 1.

In summary, the four components in Eq. (12) are:

- **Prior** $P(\theta)$: Belief about θ before observing data.
- **Likelihood** $P(D|\theta)$: How probable the observed data are for each possible θ.
- **Posterior** $P(\theta|D)$: Updated belief about θ after incorporating the data.
- **Evidence** $P(D)$: Normalizing constant ensuring the posterior integrates to 1:

$$p(D) = \int p(D|\theta)\, p(\theta)\, \mathrm{d}\theta. \tag{13}$$

  In practice, computing $p(D)$ involves an often intractable integral over all possible θ, which introduces the main computational bottleneck.

With Bayesian inference, instead of obtaining single point estimates of θ, we obtain a full distribution over θ (the posterior). This distribution can be used for prediction just like point

estimates, while also providing a measure of uncertainty, though it generally introduces additional computational cost.

2.2.1 The Recursion

1. The Initial Setup

- Define a **Prior** $p(\theta)$ (our belief before seeing any data).
- Define a **Likelihood** $p(D|\theta)$ (our model of how data is generated).

2. The Iterative Loop

- Observe new data D_n.
- Compute the **Evidence** (Marginal Likelihood) $p(D_n)$.
- Compute the **Posterior** $p(\theta|D_n)$ using Bayes' Theorem.
- **Update:** The current posterior becomes the *prior* for the next step.

3. The n-th Recursive Step

$$\underbrace{p(\theta \mid D_n, D_{n-1}, \ldots, D_0)}_{\text{New Posterior}} = \frac{\overbrace{p(D_n \mid \theta)}^{\text{Likelihood}} \cdot \overbrace{p(\theta \mid D_{n-1}, \ldots, D_0)}^{\text{Prior (Previous Posterior)}}}{\underbrace{p(D_n \mid D_{n-1}, \ldots, D_0)}_{\text{Evidence}}}$$

As n increases, our uncertainty in θ typically shrinks.

2.2.2 Simple Application: Naive Bayes Classifiers

A straightforward application of Bayesian inference is the *naive Bayes classifier*. It is termed "naive" because it assumes that all features in the dataset are **independent** (uncorrelated). In this setting, the model replaces the parameter θ with the class labels Y, e.g., $Y = 0, 1$, while the feature values X are assumed to be discrete.

For continuous features, we must assume a parametric form for the likelihood $p(X|Y)$; for example, a Gaussian distribution is used in Gaussian naive Bayes.

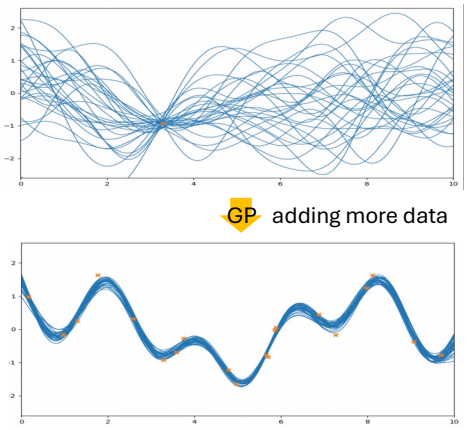
Since the independence assumption is rarely satisfied in chemistry datasets, naive Bayes classifiers are generally of limited practical use in these applications.

2.3 Gaussian Process (GP)

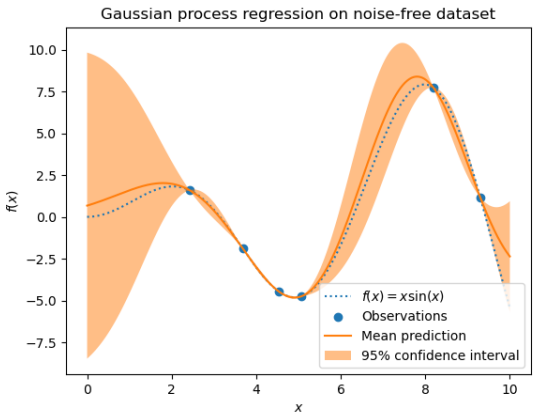
Gaussian Processes (GPs) are widely used in supervised learning and have many applications in chemical science. The key idea is that, instead of defining a distribution over model parameters

(e.g., θ), a GP defines a *distribution over functions*. Given observed data, the posterior distribution over functions is updated, and uncertainty decreases in regions where data are available.

An illustration of the learning objective of a GP is shown in Fig. 3 (a).



(a) The learning objective of GP



(b) An example of GP with data points.

Figure 3: (a) The learning objective of Gaussian process. Adapted from Ref.[1]. (b) An example of GP with 6 data points from `scikit-learn` website.

2.3.1 The Workflow

Initially, the distribution of $f(\mathbf{x})$ is broad, reflecting high uncertainty about the function. As we add observed data points $\mathbf{x}_i$, the distribution of $f(\mathbf{x}_i)$ becomes sharply peaked: zero variance for noise-free data or small variance if the observations contain noise. Importantly, adding each new data point also reduces uncertainty in nearby inputs, effectively "squeezing" the distribution in regions supported by the data. An illustration is shown in Fig. 3 (b).

We assume that the function values follow a Gaussian distribution:

$$f(\mathbf{x}) \sim \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x})), \tag{14}$$

where $\mathbf{x}$ is the input feature vector, $\mu(\mathbf{x})$ is the mean function, and $\sigma^2(\mathbf{x})$ is the variance function.

Both $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$ are determined using a **kernel function** $k(\mathbf{x}, \mathbf{x}')$, which encodes similarity between data points.

2.3.2 The Kernel

The purpose of the kernel is to relate the distributions of $f(\mathbf{x})$ at different input points. In other words, knowing $f(\mathbf{x}_i)$ helps reduce the uncertainty in $f(\mathbf{x}_j)$ for other points $\mathbf{x}_j$.

The kernel encodes the similarity between data points and is crucial for capturing correlations between function values. For n observed points, the kernel defines an $(n \times n)$ matrix K, where each entry $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ represents the covariance between $f(\mathbf{x}_i)$ and $f(\mathbf{x}_j)$. Therefore, the kernel implicitly models correlations in function space rather than directly among the input features.

When predicting the distribution of $f(\mathbf{x}_*)$ at a new input $\mathbf{x}_*$, we compute the kernel vector $\mathbf{k}_*$, containing the kernel values between $\mathbf{x}_*$ and all observed points. The predictive mean $\mu(\mathbf{x}_*)$ and variance $\sigma^2(\mathbf{x}_*)$ are then obtained using both the kernel matrix K and the vector $\mathbf{k}_*$.

2.3.3 Common Kernels

Commonly used kernels include the **radial basis function (RBF)**, **Matérn**, and **rational quadratic** kernels. In chemistry, the RBF and Matérn kernels are frequently applied for reaction prediction and experimental design. These kernels are readily available in `scikit-learn`.

Example: A Simple Gaussian Process

We illustrate the GP workflow for a single-feature function. Suppose we have two observed data points:

- $x_1 = 0, \quad y_1 = f(x_1) = 0$
- $x_2 = 2, \quad y_2 = f(x_2) = 0.5$

We want to evaluate the distribution of $f(x)$ at a new point, $x_* = 1$.
Kernel. We use an RBF kernel:

$$k(x, x') = \exp\Big(-\frac{(x - x')^2}{2}\Big).$$

Kernel matrix. For the observed points, the kernel matrix K is

$$K = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) \\ k(x_2, x_1) & k(x_2, x_2) \end{pmatrix} = \begin{pmatrix} 1 & e^{-2} \\ e^{-2} & 1 \end{pmatrix}.$$

Cross-covariance vector. The kernel vector between x_* and the observed points is

$$\mathbf{k}_* = \begin{pmatrix} k(x_1, x_*) \\ k(x_2, x_*) \end{pmatrix} = \begin{pmatrix} e^{-0.5} \\ e^{-0.5} \end{pmatrix}.$$

Predictive distribution. The predictive mean and variance of $f(x_*)$ are

$$\mu_* = \mathbf{k}_*^\top K^{-1}\mathbf{y} \approx 0.27,$$
$$\sigma_*^2 = k(x_*, x_*) - \mathbf{k}_*^\top K^{-1}\mathbf{k}_* \approx 0.35.$$

Therefore, the predictive distribution at $x_* = 1$ is

$$f(1) \sim \mathcal{N}(0.27, 0.35).$$

Similarly, we can compute the predictive distribution at other points using the same procedure.

2.3.4 Connecting Back to Bayesian Inference

It may not be immediately obvious, but GP is a form of Bayesian inference. Instead of defining a prior over model parameters θ, a GP defines a *prior over functions*, determined by the kernel function. In this sense, the kernel encodes our prior assumptions about the smoothness and correlations of the function.

The observed function values $f(x_i)$ play the role of the likelihood, while the predictive distribution over unknown points corresponds to the posterior. In other words, once we observe data, the GP updates the prior into a posterior distribution over functions.

Conceptually, the workflow can be summarized as:

Prior assumptions (kernel) + observed data $\longrightarrow$ Posterior predictive distribution of $f(x)$ at new points

3 Lab 5

In this lab, we will apply the statistical methods we learned to chemical problems. We will use a new python scikit-learn: `https://scikit-learn.org/stable/`, used as `sklearn`, which provides modules for simple predictive tasks.

Installation using `pip`:

```
pip install scikit-learn
```

Import and use it with the name `sklearn`

```
import sklearn
```

Lab link:

5-1 Linear models:

https://colab.research.google.com/drive/1B1OvZqaS0ycjTzyZ5O3x99JbSCVLLFto?usp=sharing

5-2 Gaussian Process:

https://colab.research.google.com/drive/1pMNDAxxH8G8AHdHf7Km-dvi51ODmgasp?usp=sharing

Appendix

A Non-Linear Methods

In this section, we introduce three widely known non-linear methods: decision trees (and their ensemble variant, random forests), k-nearest neighbors (kNN), and support vector machines (SVM).

While kNN and SVM are less commonly used in modern chemistry applications, we will cover them briefly so that you are familiar with the terminology and concepts.

A.1 Decision Trees and Random Forest

A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree structure, which consists of *a root node*, *branches*, *internal nodes* and *leaf nodes*. A general form of decision tree is shown in Fig. 4 (a).

We pick one of the features as the root node, the rest of the features are internal nodes. From the root node or each internal node, two branches are generated and connected to an internal node or a leaf node. The leaf nodes are the predicted y values.

To understand how decision tree works, let's look at an example.

Example: Solubility with decision tree

We would like to decide the solubility of a compound, and the target variable is

$$y = \begin{cases} 1 & \text{high solubility} \\ 0 & \text{low solubility} \end{cases}$$

The following features ($\mathbf{x}$) are used,

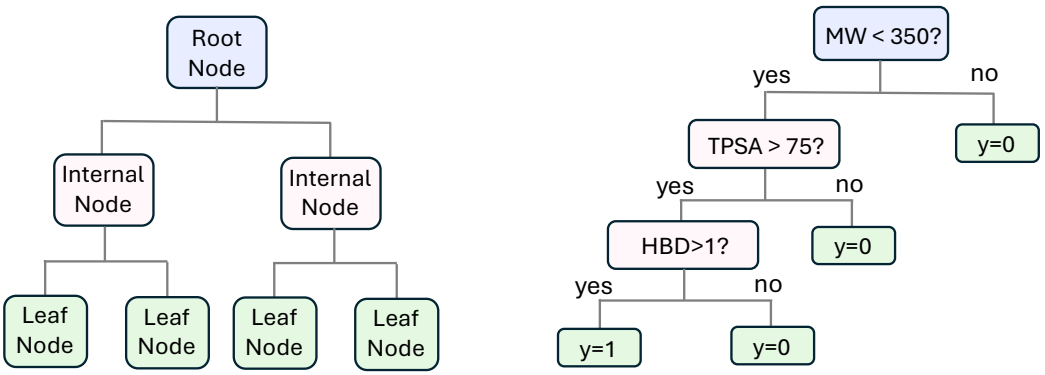
- Molecular weight (MW)
- Topological polar surface area (TPSA)
- Number of H-bond donors (HBD)

A corresponding decision tree is shown in Fig. 4 (b).

A.1.1 Training a Decision Tree

Decision trees are not continuous models and do not have a fixed functional form like regression models. How, then, do we "train" a decision tree?

We optimize the tree structure using a **greedy**, discrete algorithm. A greedy algorithm makes the best choice at each step, hoping to achieve a globally good solution. Training involves three key



(a) A general form of decision tree (b) Decision tree for solubility

Figure 4: Decision Tree

decisions at each node:

- **Split rule:** Which feature and threshold to split on. For example, should we split on molecular weight (MW) or TPSA, and if MW, what threshold t should define MW $< t$? The pseudocode is shown in Algorithm 1.
- **Tree structure:** Whether a node should become an internal node (continue splitting) or a leaf node (stop splitting).
- **Leaf prediction:** The value assigned to a leaf node for making predictions (e.g., the majority class for classification).

These decisions are evaluated using a cost function based on either **Gini impurity** or **entropy**:

$$\begin{aligned} \text{Gini impurity:} \quad & G = 1 - \sum_{i=1}^{C} p_i^2 \\ \text{Entropy:} \quad & S = -\sum_{i=1}^{C} p_i \log_2 p_i \end{aligned} \tag{15}$$

Here, C is the number of classes, and p_i is the fraction of samples in the current node belonging to class i. Note that p_i is computed using the subset of samples that reach that node, not the entire dataset.

We first explain the split rule of a root or internal node in Alg. 1.

Algorithm 1 Split Rule for Decision Tree

for $f \in$ features **do**
  for $t \in$ {possible threshold values for f} **do**
    Split the data into left ($f < t$) and right ($f \geq t$) subsets.
    Compute the weighted impurity $I(f, t)$.
  end for
end for
Choose the (f, t) pair with the lowest $I(f, t)$ as the split criterion for this node.

The cost function in decision tree classifiers is typically the **weighted impurity**:

$$I(f, t) = \frac{n_l}{n} G_l + \frac{n_r}{n} G_r, \tag{16}$$

where n_l and n_r are the sizes of the left and right subsets, and G is the **Gini impurity**:

$$G = 1 - \sum_{i=1}^{C} p_i^2, \tag{17}$$

where C is the number of classes and p_i is the fraction of samples in class i in the subset. For a binary classification ($y = 0$ or $y = 1$), $C = 2$.

Example: Evaluating the Weighted Impurity

Suppose we have 10 molecules and split them into a left subset of 4 molecules and a right subset of 6 molecules:

- Left subset: 3 soluble and 1 insoluble $\rightarrow G_l = 1 - (0.75^2 + 0.25^2) = 0.375$
- Right subset: 3 soluble and 3 insoluble $\rightarrow G_r = 1 - (0.5^2 + 0.5^2) = 0.5$

The weighted impurity is then

$$I = \frac{4}{10} \cdot 0.375 + \frac{6}{10} \cdot 0.5 = 0.45.$$

Next, we determine whether a node becomes a leaf node. A node becomes a **leaf** if any of the following criteria are met:

- **Node is pure:** All samples belong to the same class ($G = 0$).
- **Too few samples:** The node cannot be split reliably.
- **Maximum depth reached:** The tree depth exceeds a predefined limit.

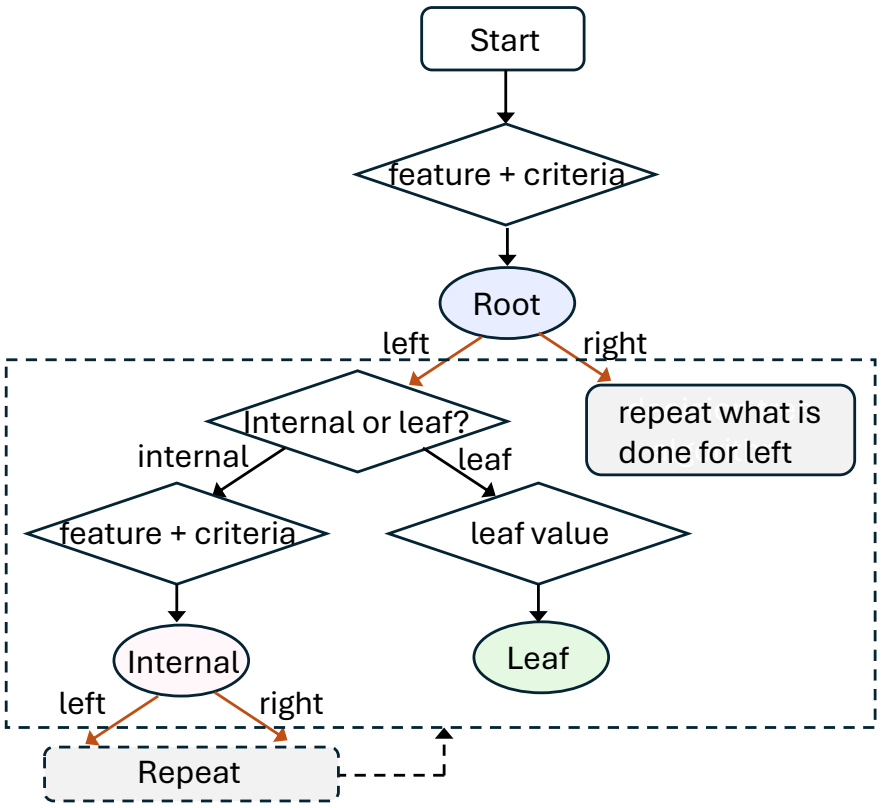


Figure 5: Flow chart to create a decision tree.

- **Minimum impurity decrease:** The split does not reduce weighted impurity sufficiently.

The *impurity decrease* from a split is computed as

$$\Delta G = G_p - \left(\frac{n_l}{n_p} G_l + \frac{n_r}{n_p} G_r \right), \tag{18}$$

where the subscript p refers to the parent node. If ΔG is below a threshold, the node becomes a leaf.

If none of the leaf criteria are met, the node becomes an **internal node** and is split according to the optimal (f, t) pair.

The value of a leaf node is assigned according to the **majority class** of samples in that branch. Finally, a flowchart illustrating the decision tree construction workflow is shown in Fig. 5.

A.1.2 Random Forest

A **random forest** is an ensemble of decision trees. Instead of relying on a single tree which may overfit the training data, we train many trees and aggregate their predictions.

Key features of a random forest:

- Each tree is trained on a *random subset of the data with replacement* (bootstrap sampling), ensuring that trees are decorrelated.
- At each node, only a *random subset of features* is considered for splitting.

The final prediction is obtained by *majority vote* across all trees for classification, or by averaging for regression.

Bootstrap sampling: After selecting a sample, it is returned to the pool before selecting the next one. As a result, some data points may appear multiple times in the subset, while others may not appear at all. This ensures that each tree sees a slightly different dataset, helping reduce correlation among trees.

A.2 k-Nearest Neighbors (kNN)

The k-nearest neighbors (kNN) algorithm is a simple, non-parametric method for both **classification** and **regression**. Unlike models such as linear regression or random forests, kNN does not learn an explicit model during training. Instead, it makes predictions directly based on the training data.

Given a new input $\mathbf{x}_{\text{new}}$, the algorithm identifies the k training samples that are *closest* to it and predicts the target value y_{new} based on these neighbors. The notion of *closeness* is defined by a distance metric, most commonly the Euclidean distance.

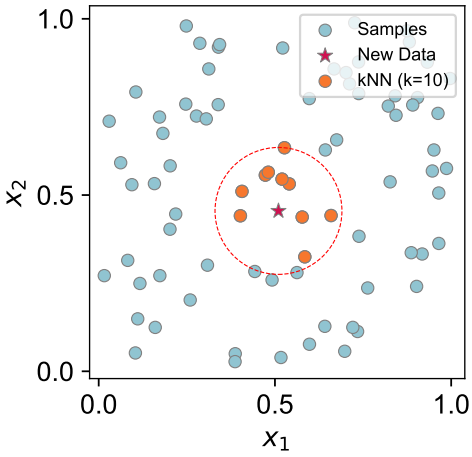


Figure 6: kNN for data with 2 features, with $k = 10$, using Euclidean distance.

The kNN algorithm proceeds as follows for a new input $\mathbf{x}_{\text{new}}$:

1. Compute the distance between $\mathbf{x}_{\text{new}}$ and all training samples.
2. Select the k closest neighbors.
3. Determine the prediction y_{new}:
    - **Classification:** Assign the class label by **majority vote** among the neighbors.
    - **Regression:** Assign the average target value of the neighbors.

An illustration of kNN with two features is shown in Fig. 6.

The kNN method is often referred to as a *lazy learning* algorithm because it does not perform explicit optimization during training. However, it can be computationally expensive at prediction time, since distances to all training samples must be evaluated. Its performance is also sensitive

to feature scaling, irrelevant or dominant features, and the choice of hyperparameters such as the number of neighbors k and the distance metric.

A.3 Support Vector Machines (SVM)

Support vector machines (SVMs) are supervised learning methods used for both **classification** and **regression**. They are particularly effective in high-dimensional feature spaces and in cases where the data is not linearly separable.

The core idea of SVM classification is to find a hyperplane that best separates the classes. The optimal hyperplane is defined as the one that maximizes the *margin*, i.e., the distance between the hyperplane and the closest data points from each class, known as *support vectors*.

SVMs can construct non-linear decision boundaries using the **kernel trick**, which implicitly maps the input data into a higher-dimensional feature space where a linear separator exists. For example, a transformation such as $x \rightarrow z = x^2$ can convert a non-linearly separable problem in x into a linearly separable one in z.

SVMs can also be applied to regression tasks, known as *support vector regression (SVR)*. In practice, however, SVR is less commonly used due to its limited scalability and relatively complex interpretation compared to other regression methods.

B Linear Model Scores

Linear regression scores:

1. R^2 **score** (also called the coefficient of determination).

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i^p - y_i^0)^2}{\sum_{i=1}^{n}(y_i^0 - \bar{y}^0)^2}$$

   where y^p is predicted value, and y^0 is the target value.
   Higher R^2 score is better.

2. **RMSE**: Root Mean Squared Error.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i^p - y_i^0)^2}$$

   Lower RMSE is better.

3. **MAE**: Mean Absolute Error.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i^p - y_i^0|$$

   Lower MAE is better.

Logistic regression scores:

1. **Accuracy.** Fraction of correct predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

   Higher accuracy is better.

2. **Precision.** Fraction of predicted positives that are actually positive.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{TP + False Positives (FP)}}$$

   Higher precision is better.

3. **Recall (Sensitivity).** Fraction of actual positives correctly predicted.

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{TP + False Negatives (FN)}}$$

   Higher recall is better.

4. **F1 Score.** Harmonic mean of precision and recall.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision + Recall}}$$

   Higher F1 score is better.

5. **ROC AUC.** Area under the Receiver Operating Characteristic curve.

   - $1 \rightarrow$ perfect classifier
   - $0.5 \rightarrow$ random guessing

   Higher ROC AUC is better.

References

[1] Jie Wang. An intuitive tutorial to gaussian process regression. https://github.com/jwangjie/Gaussian-Process-Regression-Tutorial, 2023. GitHub repository.