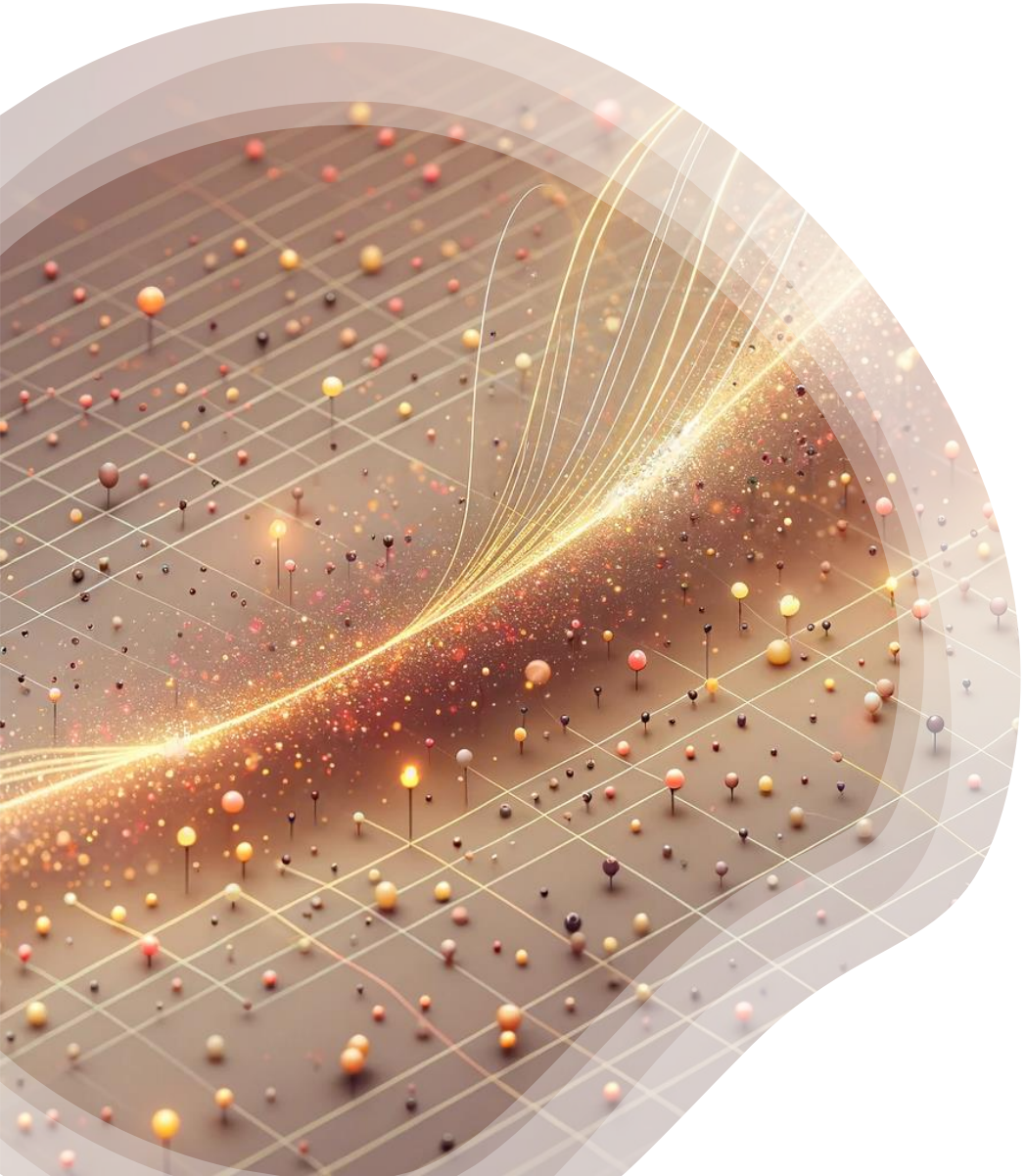
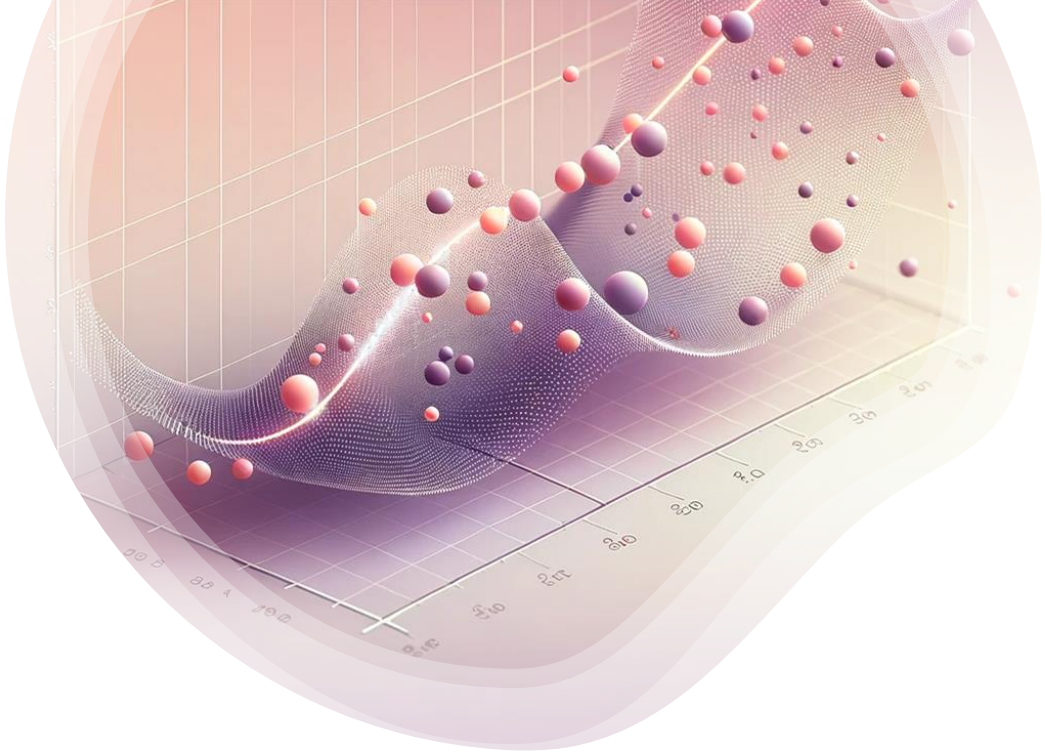
LEHIGH UNIVERSITY





Linear Regression: An Introduction to Discriminative Supervised Learning

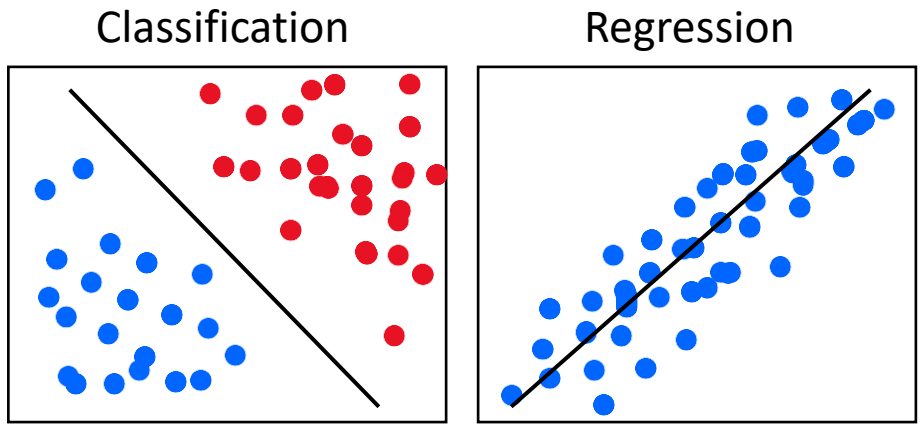
- **ISE – 364 / 464**
- **Dept. of Industrial and Systems Engineering**
- **Griffin Dean Kent**

Supervised Learning Vs. Unsupervised Learning

(A summary to set the stage)

Supervised Learning

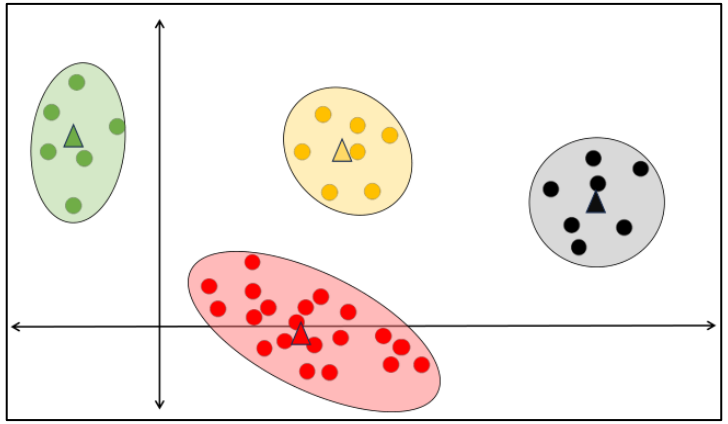
- A category of problems where one has a set of datapoints that have features along with corresponding target labels.
- The goal of these problems is to "learn" the best model that can accurately determine the target class label (for a categorical target) or the target numerical value (for a numerical target) given the feature vectors of the dataset.



Supervised Learning

Unsupervised Learning

- A category of problems where one has a set of datapoints that have only features and does not have any corresponding target labels.
- The goal of these problems is to "learn" some overall patterns that are present in the features of the dataset such as natural groupings or similarities amongst the datapoints.



Unsupervised Learning

Supervised Learning: Discriminative Vs. Generative Models

Discriminative Learning

- Models that directly aim at learning (or approximating) the **posterior probability distribution** $\mathbb{P}(Y|X)$.
- The posterior distribution $\mathbb{P}(Y|X)$ is the distribution of the target variable Y conditioned on observing a set of data features X.
- This distribution is given the name "posterior" distribution in reference to the probability of the target variable Y "post" (or after) observing the data X.
- Alternatively, we refer to $\mathbb{P}(Y)$ as the **prior distribution** since it is meant to represent our knowledge or belief in an outcome of interest Y before observing any data X, hence the name "prior".

Generative Learning

- Models that *still* predict posterior probabilities, but instead of learning the posterior distribution directly, generative models focus on learning the **underlying joint probability distribution** $\mathbb{P}(X, Y)$, after which **Baye's Theorem** can be applied to "**generate**" probabilities from the posterior distribution.
- Once the joint probability distribution $\mathbb{P}(X, Y)$ is learned, using Baye's Theorem, posterior probabilities are generated according to the equation

$$P(Y|X) = \frac{P(Y)P(X|Y)}{P(X)}.$$

Supervised Learning: Fundamentals

Data and Population Sets

- We have a **dataset** $\mathcal{D} \coloneqq \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m}$ that consists of m "examples" (or "instances"), where (i) indexes these indices.
- Each$\left(x^{(i)}, y^{(i)}\right)$ is referred to a **feature-target pair** where $x^{(i)} \in \mathcal{X} \subseteq \mathbb{R}^n$ and $y^{(i)} \in \mathcal{Y} \subseteq \mathbb{R}$.
- The set $\mathcal{X}$ is known as the **Domain Set** (or the Feature Space) and $\mathcal{Y}$ is known as the **Label Set**.
- Together, the $\mathcal{X}$ and $\mathcal{Y}$ sets can be referred to as the "**population sets**" as they are where our examples are drawn from to form the dataset $\mathcal{D}$.

Underlying Unknown Distribution

We assume that the examples in our dataset are drawn from a "**hidden**" (**unknown**) **underlying distribution**, denoted by $\mathcal{P}(x, y)$ (only Laplace's Demon has knowledge of this distribution). This is the joint probability distribution over the feature-target pairs (x, y) and can be thought of as what is used to "generate" the data.

- If this distribution was known, it would simplify all machine learning problems.
- If $\mathcal{P}(x, y)$ was known, we would have **access to the optimal model** h^*, there would be **no concerns of overfitting** or **underfitting**, we would be able to **compute the expected risk <u>exactly</u>** and not have to rely on approximating it via empirical risk along with validating techniques, etc.

Supervised Learning: Fundamentals

The Model and the Hypothesis Space

- In **supervised learning**, the goal is to learn a **prediction rule** $h: \mathcal{X} \to \mathcal{Y}$ (also known as the **model** or the **hypothesis**) from a set $\mathcal{H}$ of possible models, i.e., $h \in \mathcal{H}$, which is called the "**hypothesis space**".
- The model h is **optimized** over a set of parameters $\theta \in \mathbb{R}^n$ (the learning is the process of determining what the optimal set of parameters θ^* that will yield the "best" model). As such, we may sometimes denote the model by h_θ.

The Loss Function

- The **loss function** $J: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a metric that **measures the discrepancy** between the true label $y^{(i)}$ for an example i and the predicted label $h_\theta\left(x^{(i)}\right)$.
- In **regression problems**, the loss function typically measures the average **"distance"** between true labels and predicted labels.
- In **classification problems**, the loss function typically measures the proportion of **misclassified** examples by the model.

Supervised Learning: Fundamentals

The Expected Risk

Otherwise known as the **Total Risk** (or the **Population Risk**), the Expected Risk, denoted $\mathcal{R}(\theta)$, is the expected value of the loss function taken over the joint distribution $\mathcal{P}(x, y)$:

$$\mathcal{R}(\theta) := \mathbb{E}_{(x,y)\sim\mathcal{P}}[J(h_\theta(x), y)] = \int_{X\times Y} J(h_\theta(x), y)d\mathcal{P}(x, y)\,.$$

The Empirical Risk

As mentioned previously, we **do not typically have access** to the **underlying distribution** $\mathcal{P}(x, y)$. As a result, we instead utilize a surrogate function as an **approximation of the expected risk**: the **empirical risk**, denoted $\hat{R}_m$, which is the average loss over the training dataset $\mathcal{D}$. This is written as

$$\hat{R}_m(\theta) = \frac{1}{m}\sum_{i=1}^{m} J\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right).$$

Supervised Learning: The Goal

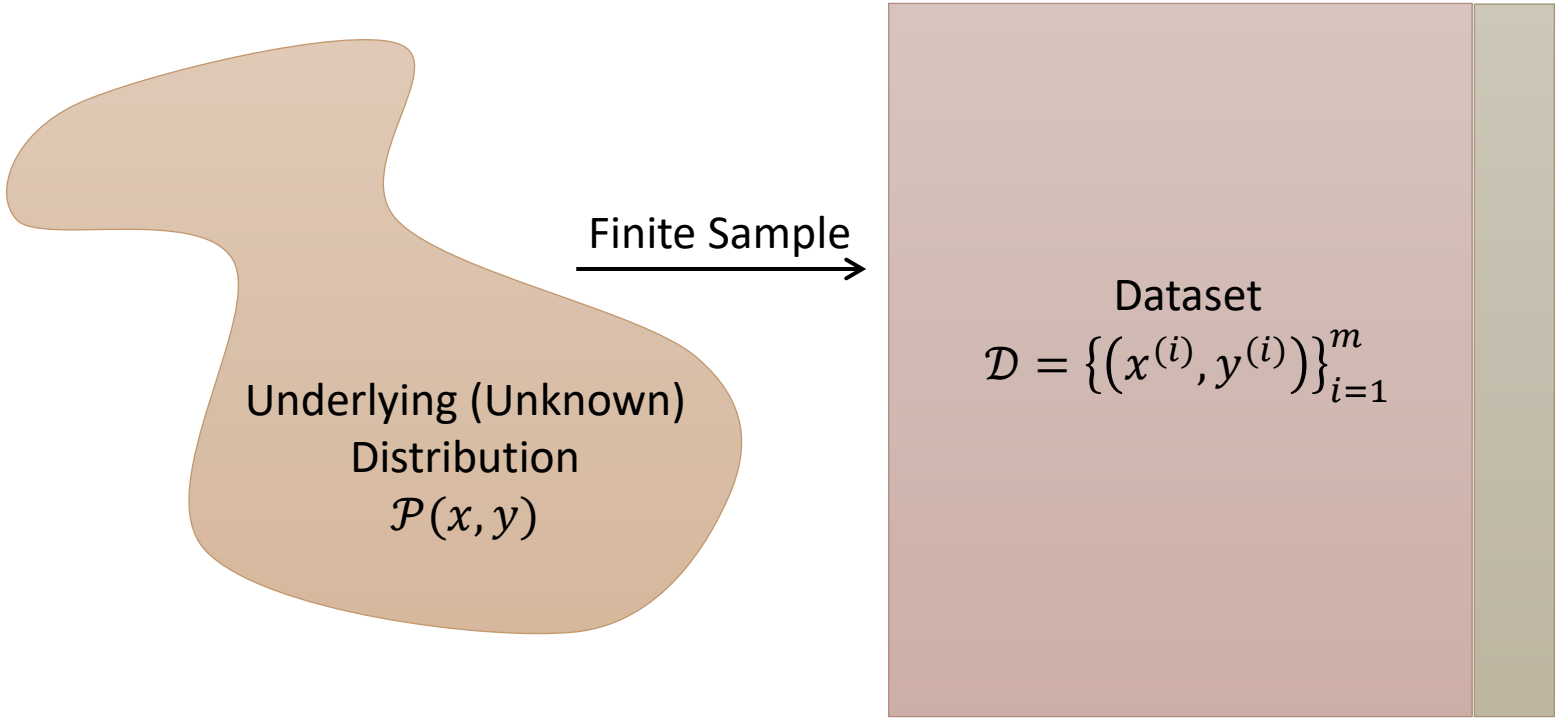
The Goal of Supervised Learning

The ultimate goal of supervised learning is to learn a model h_θ that minimizes the expected risk $R(\theta)$ with the explicit desire to either **discriminate** between target classes (via either discriminative or generative models) to predict posterior probabilities $\mathbb{P}(Y|X)$ or to **generate** new data according to the underlying distribution $\mathcal{P}(X, Y)$.

However, since we do not have complete knowledge of the underlying distribution $\mathcal{P}$, we go about accomplishing this by taking a sample from the distribution $\mathcal{P}$ to obtain a training dataset $\mathcal{D}$. We then learn a model h_θ that minimizes the empirical risk $\hat{R}_m(\theta)$ to approximate the expected risk.

Further, such a learning process involves first choosing a space of model classes $\mathcal{H}$ and then selecting the model h_θ (by choosing the parameters θ) that minimizes the error.

Obtaining a Representative Sample of $\mathcal{P}$



- Choosing a **representative sample** of the underlying distribution $\mathcal{P}$ is a **VERY important** part of the ML process.
- Ultimately, we want to learn a model that correctly identifies patterns that are **representative of reality**.
- However, if our dataset is not representative of the underlying distribution (i.e., our dataset is biased), then it does not reflect reality, and there's no way that our model will reflect reality either (i.e., a biased model).
- Hypothetically, this problem could be resolved with correct **sampling techniques** and **statistical acumen**.
- However, this has spawned its own subfield of research in ML, known as **Fair Machine Learning**, which aims to remedy this problem at the algorithm level.

General Notation

Design Matrix and Target Vector

- We have a finite dataset $\mathcal{D} \coloneqq \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m}$.
- Each $x^{(i)}$ and $y^{(i)}$ are realizations of the random variables $\mathcal{X}$ and $\mathcal{Y}$.
- Each training instance (i) has a feature vector $x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^n$ and a corresponding target $y^{(i)} \in \mathbb{R}$.
- For ease of notation and mathematical manipulation, we can condense all the feature vectors into a **design matrix** defined as

$$X \coloneqq \begin{bmatrix} x^{(1)T} \\ x^{(2)T} \\ \vdots \\ x^{(m)T} \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

- We can also condense all the corresponding targets into a **target vector** defined as

$$y \coloneqq \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^{m}.$$

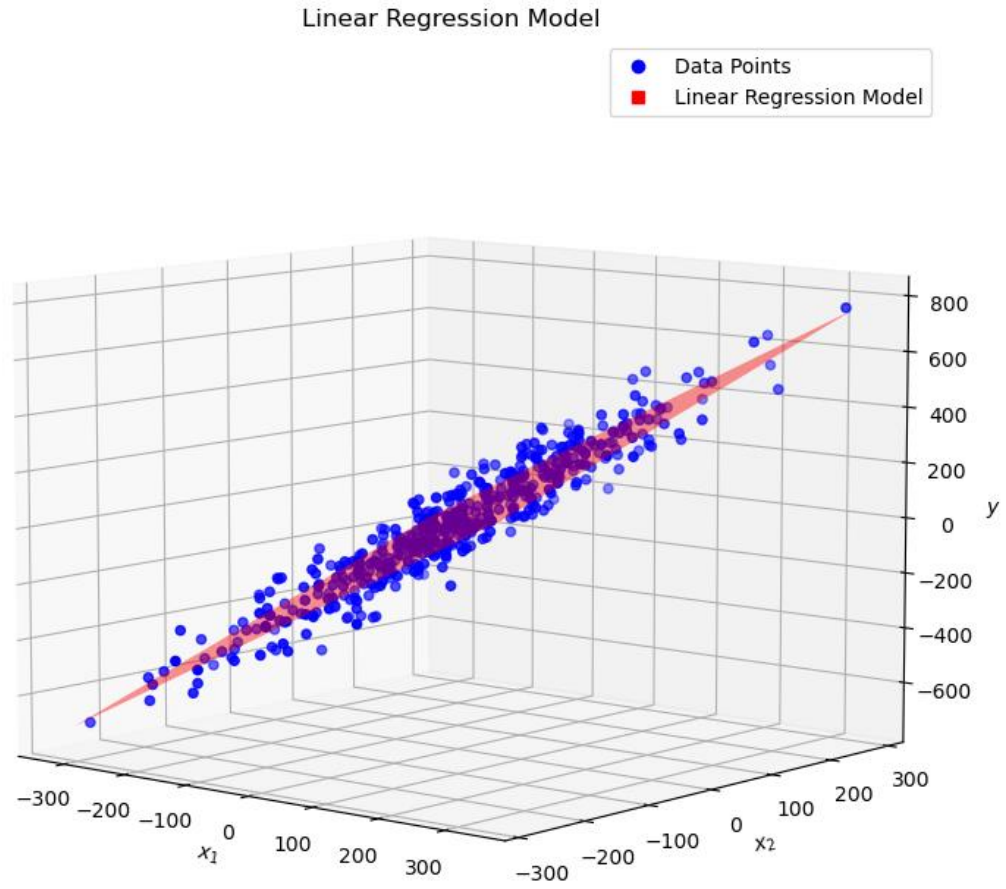
Linear Model – Hyperplane

Linear Regression Model

- Typically regarded as the simplest and most intuitive predictive model to understand (except for KNN).
- Essentially, we are trying to model a dataset with a **hyperplane**.
- We **assume** that we are dealing with **real-valued features** $x \in \mathbb{R}^n$ and a **real-valued target** $y \in \mathbb{R}$.
- Further, we **also assume** that all the feature and target vectors are **normally distributed**. Why?
- A **linear regression model** is a function $h_\theta(\cdot)$ (or $h(\cdot\,;\theta)$) that is linear in its parameters $\theta \in \mathbb{R}^{n+1}$and is defined by the following hyperplane:

$$h(x\,;\theta) \coloneqq \theta^T x = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n.$$

- In this equation, x denotes any feature vector (i.e., any row vector from the design matrix X). Further, we typically augment the x vectors by appending a 1 as the first entry, making $x \in \mathbb{R}^{n+1}$ to match the dimension of θ.



Choice of Loss Function

- How do we determine **how good** our current choice of parameters are?
- We need to decide on some sort of "**distance metric**" that measures our model's performance.
- This essentially amounts to choosing a function that measures the **error** (or **loss**, hence the name "**loss function**") of a prediction that our model makes when compared to the true actual value associated with that prediction.

Mean Absolute Error (MAE)

- To compute the loss over an entire dataset, we can simply compute the average of all m **absolute differences** via the following:

$$J(\theta\,; x, y) \coloneqq \frac{1}{m}\sum_{i=1}^{m}\left|h\left(x^{(i)}\,;\,\theta\right) - y^{(i)}\right|.$$

- This is also known as the ℓ_1-loss (after the ℓ_1-norm).

Mean Squared Error (MSE) & Root Mean Squared Error (RMSE)

- Another choice of loss function is the classic **MSE** function, and is exactly as named:

$$J(\theta\,; x, y) \coloneqq \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(h\left(x^{(i)}\,;\,\theta\right) - y^{(i)}\right)^2.$$

- Further, we can also take the square root of this equation to obtain a metric in the same units, yielding the **RMSE**:

$$J(\theta\,; x, y) \coloneqq \sqrt{\frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}\,;\,\theta) - y^{(i)})^2}.$$

- This is also known as the ℓ_2-loss (after the ℓ_2-norm) or the **Euclidean distance** between two points.

Linear Regression

- **Learning** is the process that selects θ to **minimize** the loss function J.

Linear Regression

Linear Regression is the process that selects θ to minimize the Mean Squared Error loss function via a linear model (hyperplane), i.e., linear regression solves the problem (where h is a hyperplane):

$$\theta^* = \underset{\theta \in \mathbb{R}^{n+1}}{\operatorname{argmin}} J(\theta) = \underset{\theta \in \mathbb{R}^{n+1}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \left(h\left(x^{(i)} ; \theta\right) - y^{(i)} \right)^2 .$$

- We can rewrite the MSE function in a more compact fashion by utilizing **vector-norm notation**.
- To do this, let's collect all the model's predictions into a single vector as

$$h(X ; \theta) \coloneqq \begin{bmatrix} h\left(x^{(1)} ; \theta\right) \\ h\left(x^{(2)} ; \theta\right) \\ \vdots \\ h\left(x^{(m)} ; \theta\right) \end{bmatrix} = \begin{bmatrix} \theta^T x^{(1)} \\ \theta^T x^{(2)} \\ \vdots \\ \theta^T x^{(m)} \end{bmatrix} = X\theta \in \mathbb{R}^m .$$

- Using this, we can write the **compact vectorized form** of the **MSE** as

$$J(\theta) = \frac{1}{2m} \| h(X ; \theta) - y \|_2^2 = \frac{1}{2m} \| X\theta - y \|_2^2 = \frac{1}{2m} (X\theta - y)^T (X\theta - y).$$

- Now, the most important question: how do we go about minimizing the MSE?

Training a Linear Regression Model Via Gradient Descent

- In machine learning problems in general, the **search space** for θ is likely very large or we may be dealing with very large datasets.
- Hence, we need an efficient way to find the best parameters.
- Typically, in ML we utilize some form of **gradient descent** due to its **low computational cost** (being a first-order method).
- As such, the first thing we need is an expression of the gradient itself, i.e., the collection of partial derivatives of the loss function w.r.t. each parameter θ_j for all $j \in \{0,1,2,\dots,n\}$, defined as

$$\nabla J(\theta) = \frac{\partial J(\theta)}{\partial \theta} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}.$$

- Thus, we need an expression for each $\frac{\partial J(\theta)}{\partial \theta_j}$.

Derivation of the Gradient of the MSE Loss

(for each individual $\boldsymbol{j}$ component)

$$\frac{\boldsymbol{\partial J(\theta)}}{\boldsymbol{\partial \theta_j}} = \frac{\partial}{\partial \theta_j}\left(\frac{1}{2m}\sum_{i=1}^{m}\left(h\left(x^{(i)};\,\theta\right)-y^{(i)}\right)^2\right)$$

$$= \frac{1}{2m}\frac{\partial}{\partial \theta_j}\left(\sum_{i=1}^{m}\left(h\left(x^{(i)};\,\theta\right)-y^{(i)}\right)^2\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^{(i)};\,\theta\right)-y^{(i)}\right)\frac{\partial}{\partial \theta_j}\left(h\left(x^{(i)};\,\theta\right)-y^{(i)}\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^{(i)};\,\theta\right)-y^{(i)}\right)\frac{\partial}{\partial \theta_j}\left(\theta_0+\theta_1 x_1+\cdots+\theta_n x_n-y^{(i)}\right)$$

$$= \boldsymbol{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(x^{(i)};\,\theta\right)-y^{(i)}\right)x_j^{(i)}}.$$

Derivation of the Gradient of the MSE Loss

(compact vector notation)

$$\boldsymbol{\nabla J(\theta)} = \frac{1}{2m}\frac{\partial}{\partial\theta}[(X\theta - y)^T(X\theta - y)]$$

$$= \frac{1}{2m}\frac{\partial}{\partial\theta}[(X^T\theta^T - y^T)(X\theta - y)]$$

$$= \frac{1}{2m}\frac{\partial}{\partial\theta}[\theta^T X^T X\theta - 2\theta^T X^T y - y^T y]$$

$$= \frac{1}{2m}\left[\frac{\partial}{\partial\theta}\theta^T X^T X\theta - \frac{\partial}{\partial\theta}2\theta^T X^T y - \frac{\partial}{\partial\theta}y^T y\right]$$

$$= \frac{1}{2m}[(X^T X + (X^T X)^T)\theta - 2X^T y]$$

$$= \frac{1}{2m}[2X^T X\theta - 2X^T y]$$

$$= \frac{1}{m}[X^T X\theta - X^T y]$$

$$= \boldsymbol{\frac{1}{m}X^T[X\theta - y]}.$$

The Normal Equations

Let's consider again the linear regression problem:

$$\theta^* = \underset{\theta \in \mathbb{R}^{n+1}}{\operatorname{argmin}} J(\theta) = \underset{\theta \in \mathbb{R}^{n+1}}{\operatorname{argmin}} \frac{1}{2m} \|h(X\,;\,\theta) - y\|_2^2 .$$

- Why are we bothering minimizing the MSE with gradient descent when it is a convex function in the case of linear regression?
- Simply set the gradient equal to 0 and solving for the optimal parameters in the following way:

$$\nabla J(\theta) = \frac{1}{m} X^T [X\theta - y] = 0$$

$$\rightarrow \boldsymbol{\theta^* = \left(X^T X\right)^{-1} X^T y}.$$

This system of linear equations is known as the **Normal Equations**.

Pros

- Closed form solution that does not require iterative optimization.
- No need for hyperparameter tuning like choosing a learning rate.

Cons

- The Inversion $\boldsymbol{\left(X^T X\right)^{-1}}$ is very computationally expensive for large datasets (instances and features). Similarly, the memory cost can be prohibitively expensive for large datasets.
- Computing the Inversion $\boldsymbol{\left(X^T X\right)^{-1}}$ can also be numerically unstable when $\boldsymbol{X^T X}$ is ill-conditioned.

A Probabilistic Interpretation of Linear Regression

What if we model the errors in predicting the ground-truth target $y^{(i)}$ by i.i.d. random variables $\epsilon^{(i)}$?

- Specifically, let's **assume** that these errors are independently distributed according to a **normal (Gaussian) distribution**, i.e., $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$.

- Now, accounting for this error, we can write the targets as $y^{(i)} = \theta^T x + \epsilon^{(i)}$. Thus, we can expect our targets to follow a linear pattern with some 0-mean Gaussin noise in the predicted value.

- It follows that our target variable itself can now be treated an i.i.d. random variable with the distribution $y^{(i)} \sim \mathcal{N}\left(\theta^T x^{(i)}, \sigma^2\right)$. Thus, using the probability density equation for a **Gaussian** random variable, the probability density function of the target $y^{(i)}$ is given by

$$f_{Y|X}\left(y^{(i)}|x^{(i)};\theta\right) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\left(y^{(i)}-\theta^T x^{(i)}\right)^2}{2\sigma^2}}.$$

Likelihood for Probabilistic Linear Regression

- We can now ask ourselves: "Given a model parameterized by θ, what is the probability (or likelihood) of observing each normally distributed target variable $y^{(i)}$ "conditioned" on observing the corresponding feature data $x^{(i)}$?"
- Given the density function of the target $y^{(i)}$, we can write the **likelihood function** of θ that corresponds to observing the m samples $\left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m}$ as the following:

$$L(\theta) = L\left(\theta; \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{m}\right) = \mathbb{P}\left(\left\{\left(y^{(i)}|x^{(i)}\right)\right\}_{i=1}^{m}; \theta\right) = \prod_{i=1}^{m} \mathbb{P}\left(\left(y^{(i)}|x^{(i)}\right); \theta\right)$$

$$= \prod_{i=1}^{m} f_{Y|X}\left(y^{(i)}|x^{(i)}; \theta\right) = \prod_{i=1}^{m} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}}$$

$$= \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^m \exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{m}\left(y^{(i)} - \theta^T x^{(i)}\right)^2\right\}.$$

Linear Regression via Maximum Likelihood

Maximum Likelihood Estimation (MLE) for θ

- We now have the **likelihood function** of θ that corresponds to observing the m samples.
- Intuitively, we can should choose the value of θ that will maximize the likelihood of observing the m samples, i.e.,

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, L(\theta)\,.$$

- However, since sums are typically easier to work with than products, we will maximize the **log of the likelihood** function (we can do this since the logarithm is a monotonically-increasing function and will not change the result of the MLE).

$$\ell(\theta) = \log L(\theta) = -m\log\left(\sigma\sqrt{2\pi}\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{m}\left(y^{(i)} - \theta^T x^{(i)}\right)^2.$$

- Do you notice anything about this new loss function?
- The optimal solution to this function is the exact same as the optimal solution to the MSE loss.
- Hence, we can train the model via the same procedures (gradient descent or by solving the normal equations).

Polynomial Regression

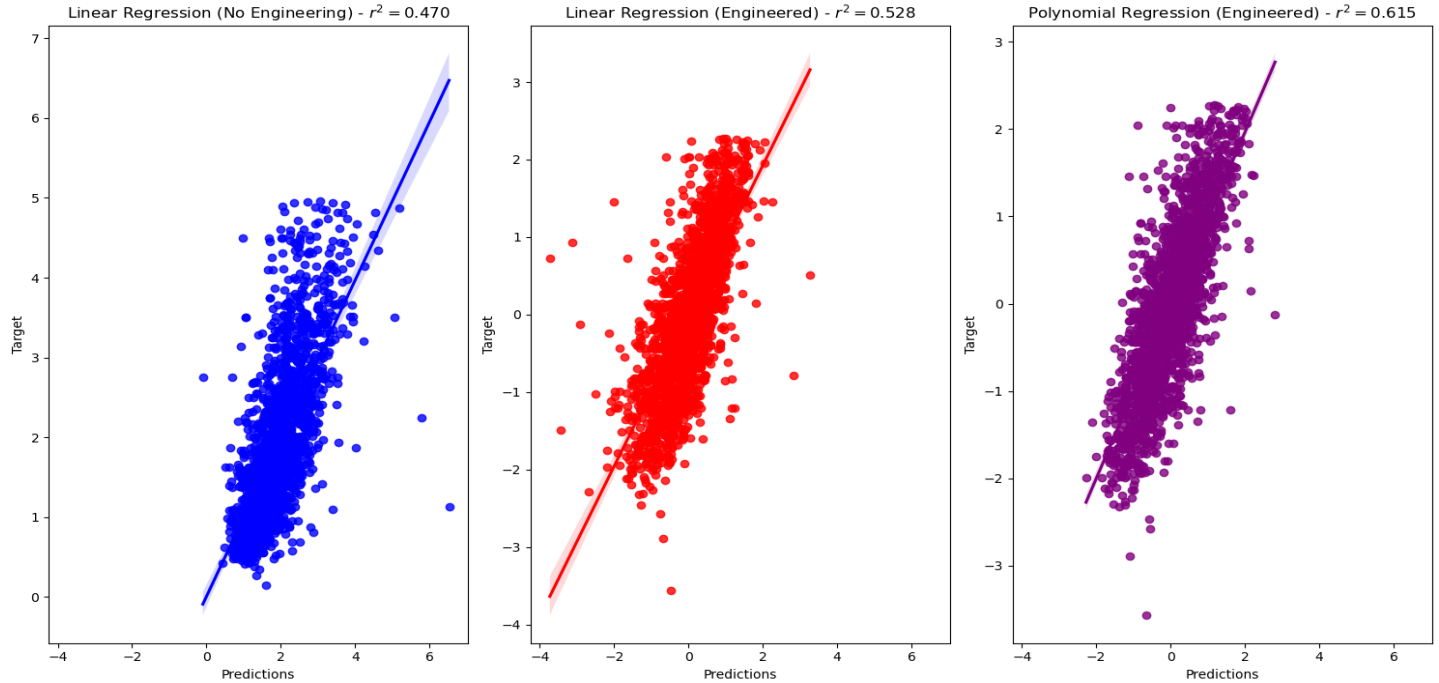
What if there are some simple nonlinear relationships that are present in the data, such as polynomial relationships? Once a linear regression model has been obtained, it can be useful to add polynomial features of the original features as well as cross-terms.

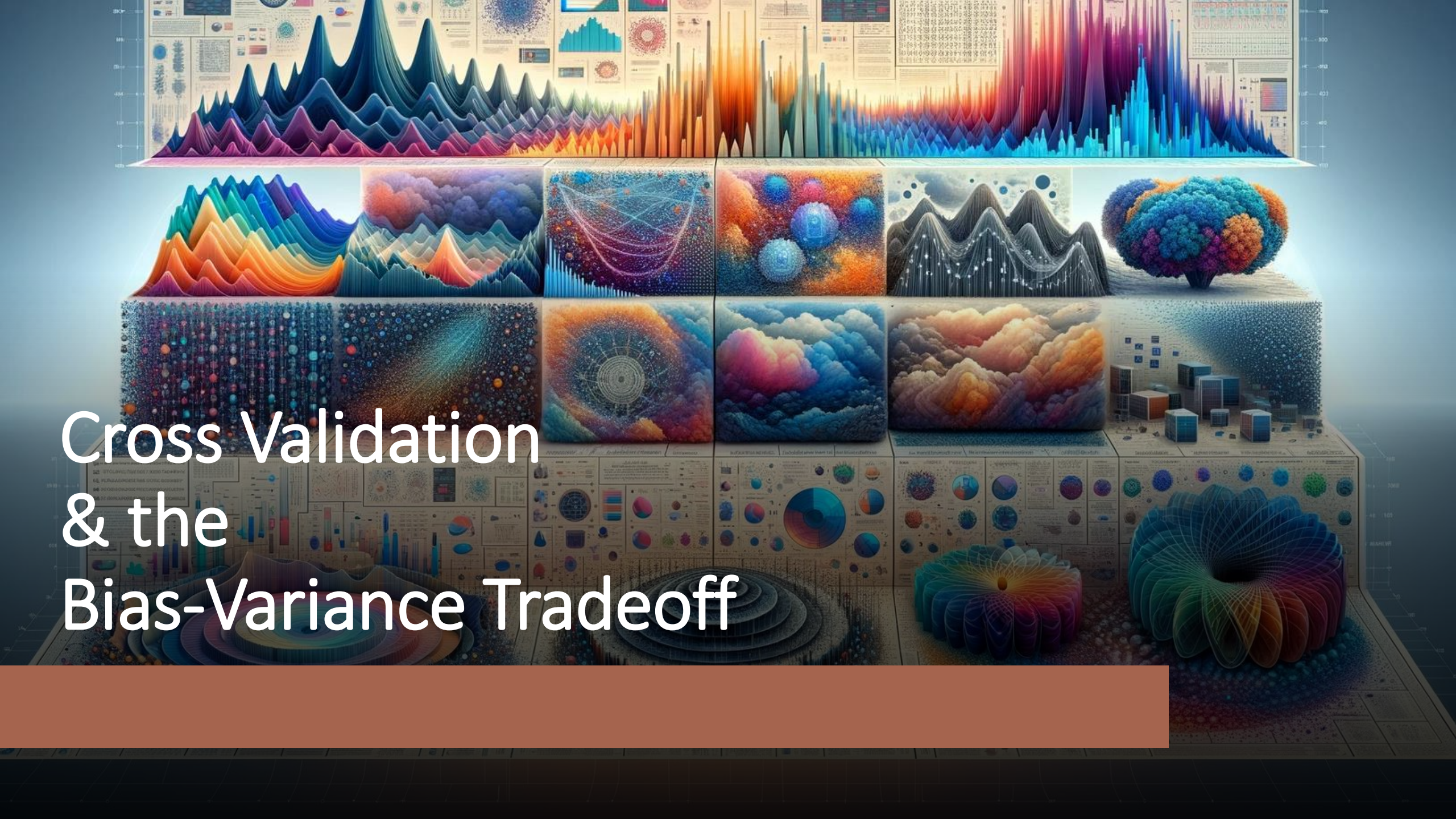
Polynomial Regression

A polynomial regression model is defined by the sum of all its terms, starting with simple linear relationships and then followed by all possible polynomial relationships up to a degree of $d \leq n$, where n is the number of features.

$$h_\theta(x) = \sum_{j=1}^{n} \theta_j x_j + \sum_{1 \leq j_1 \leq j_2 \leq n} \theta_{j_1 j_2} x_{j_1} x_{j_2} + \cdots + \sum_{1 \leq j_1 \leq \cdots \leq j_d \leq n} \theta_{j_1 j_2 \ldots j_d} x_{j_1} x_{j_2} \cdots x_{j_d} .$$

- This image is an example comparing two different linear regression models and a polynomial model on the (slightly modified) *California Housing Dataset*.
- The **blue linear model** is an example of simply applying linear regression to the un-scaled and un-normalized features and target ($r^2 \approx 0.470$).
- The **red linear model** is an example of applying linear regression when one has normalized and standardized the features and target ($r^2 \approx 0.528$).
- The **purple model** is a 3rd-degree polynomial model on the normalized and standardized data ($r^2 \approx 0.615$).





Cross Validation & the Bias-Variance Tradeoff

Artifacts of Using the Empirical Risk as a Proxy to the Expected Risk

- As we have discussed, we utilize the Empirical Risk as an approximation to the Expected Risk because we do not have knowledge of the underlying distribution $\mathcal{P}$.
- However, this approximation can create some issues if not handled carefully.
- Remember, training a model to minimize the empirical risk is training a model to minimize the prediction error on the dataset that we **currently** have. However, remember that we are using the dataset $\mathcal{D}$ to **approximate** the underlying unknown distribution $\mathcal{P}$.
- Thus, we want to ensure that the model we train can generalize to new data that comes from the distribution $\mathcal{P}$.
- There are two important concepts relating to measuring our model's generalizability.

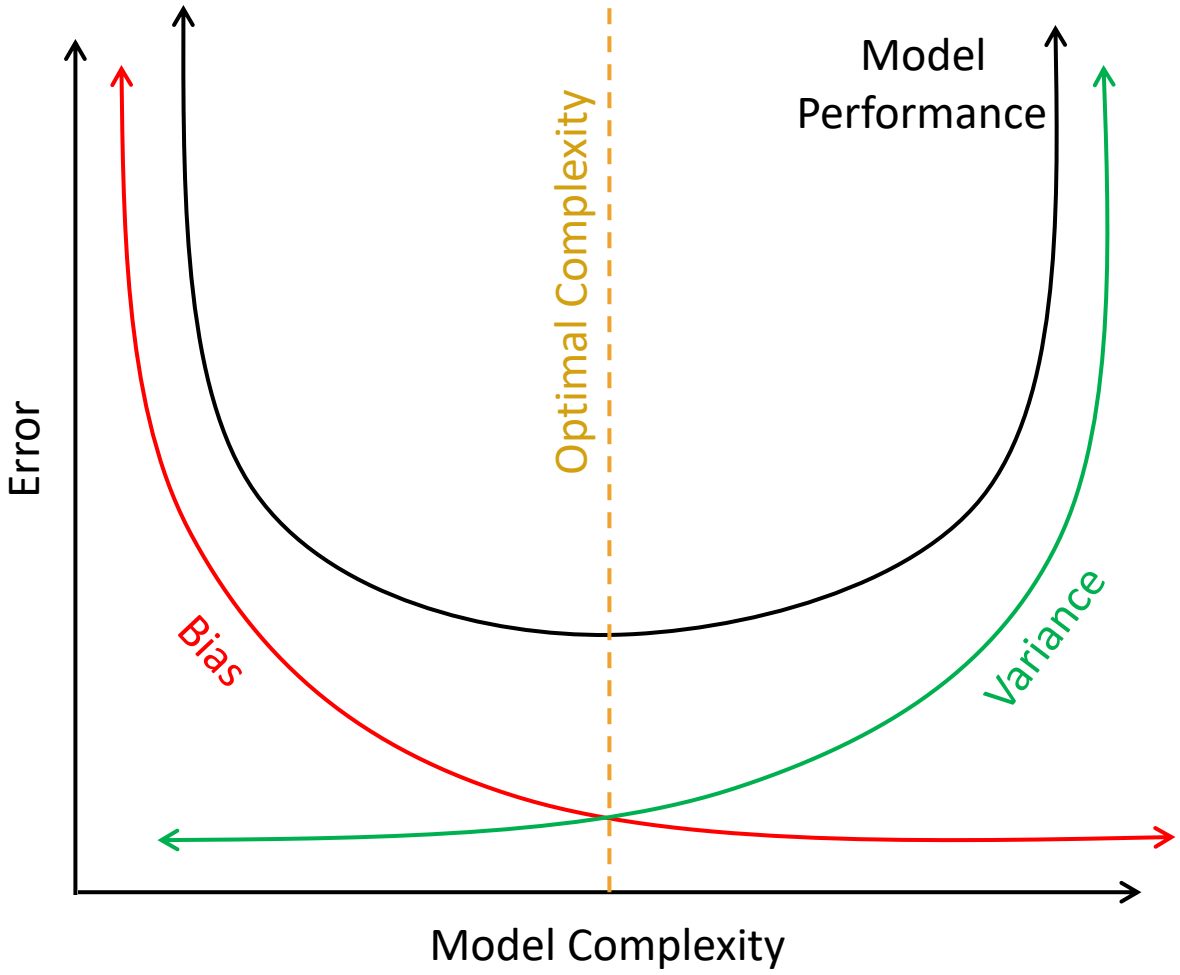
Bias

The bias of a predictive model describes the level of error in the underlying assumptions of the model itself. A high bias (also known as underfitting) indicates that the current model is not able to identify relevant patterns between features and the target output.

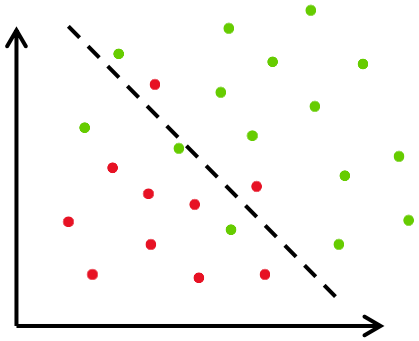
Variance

The variance of a predictive model describes the level of error due to small fluctuations in the training set. A high variance (also known as overfitting) indicates that the current model is fitting to the noise in the training dataset and is learning irrelevant patterns instead of the general overarching patterns.

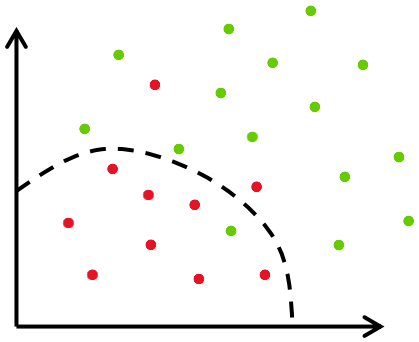
The Bias-Variance Tradeoff



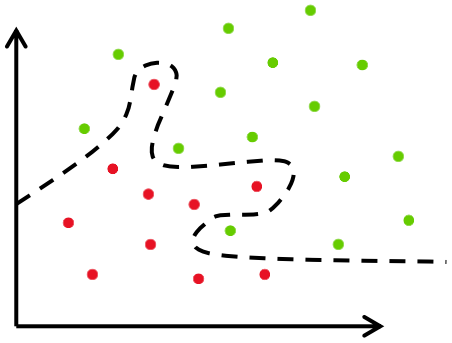
High Bias – Low Variance
(Underfitting)



Balanced Bias & Variance
(``Optimal'' Complexity)



Low Bias – High Variance
(Overfitting)



Cross Validation: Train-Test Splits

At a broad level, the solutions to the two problems of underfitting and overfitting are as follows:

- **(Underfitting)** Ensure that you chose an appropriate model for the task. Past that, just train the model more! (``Easy'')
- **(Overfitting)** Employ some form of cross-validation or regularization techniques. (``Tricky")

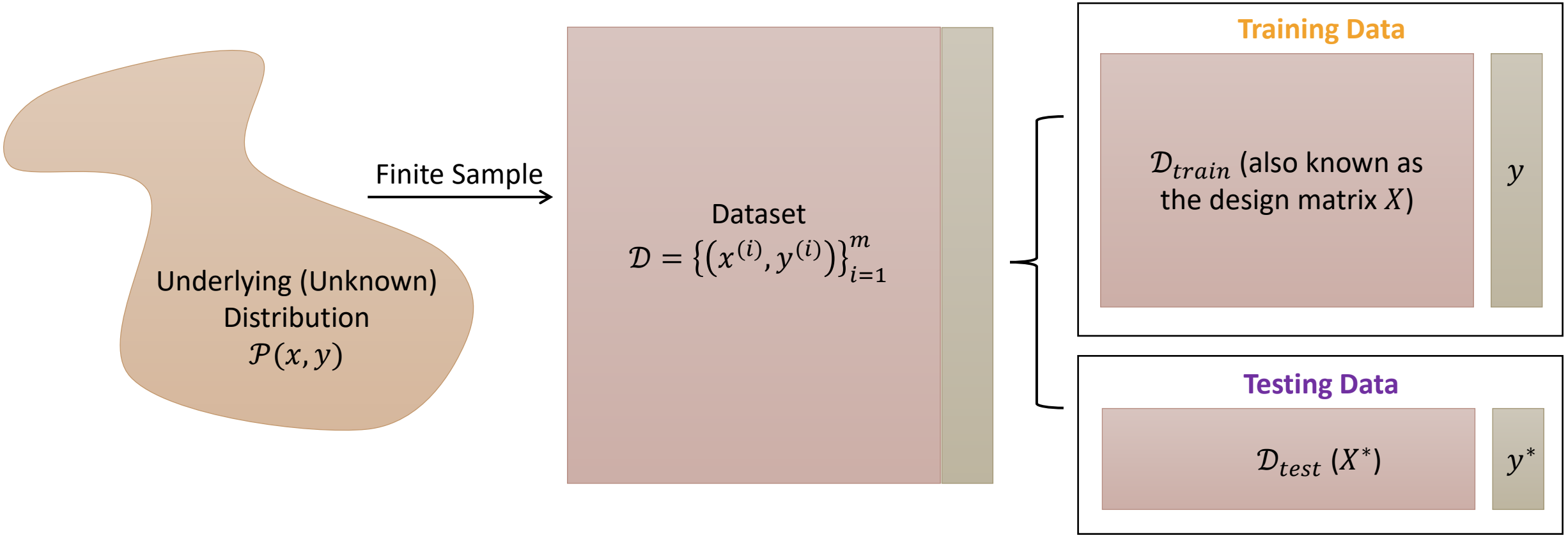
Since handling the problem of underfitting is a relatively straightforward task if one understands the different ML models and what they do, we will now just focus on how to handle cases of model **overfitting**.

Train-Test Split (Cross Validation)

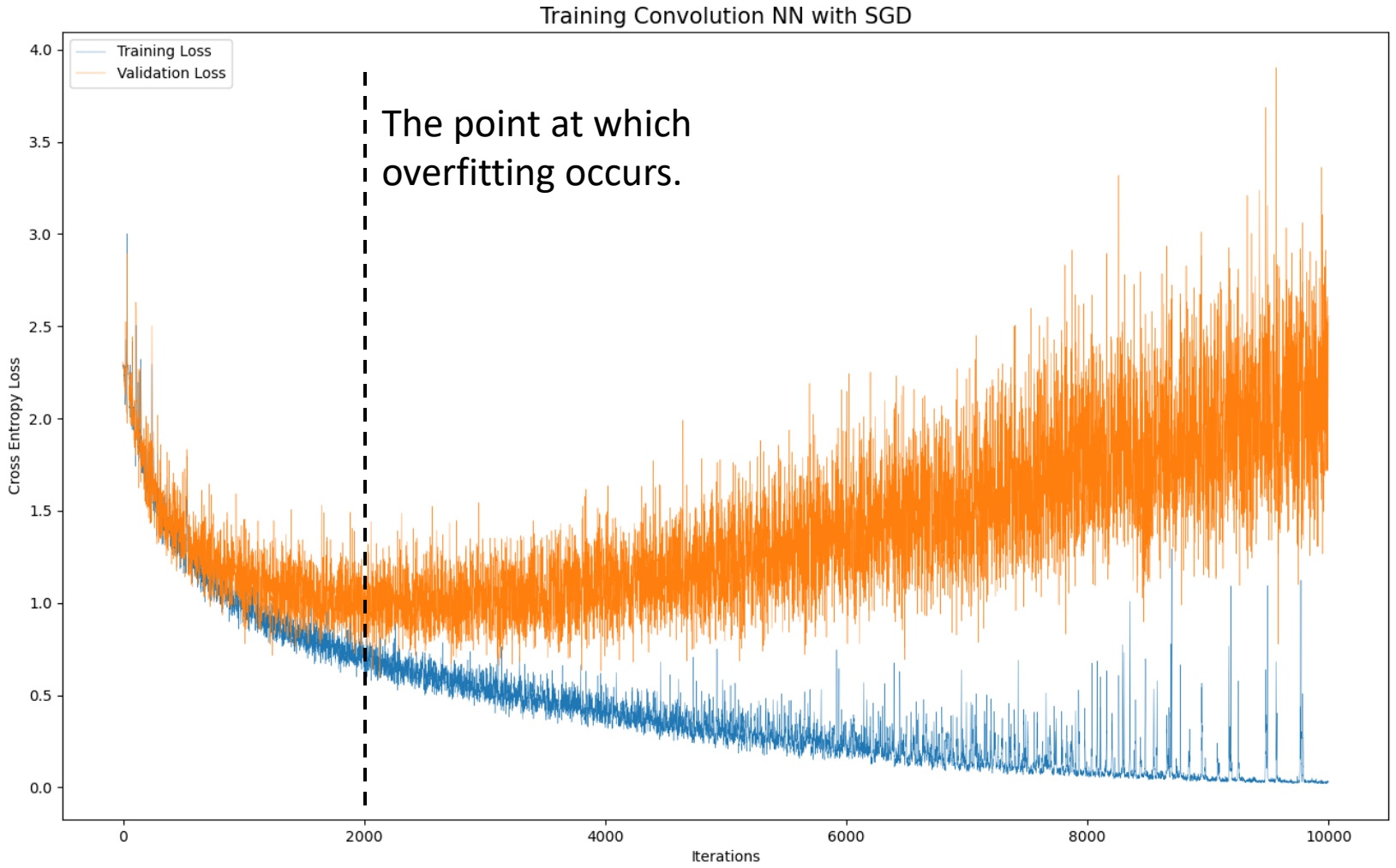
Given a dataset $\mathcal{D}$, the most classic way to help prevent overfitting is to split $\mathcal{D}$ into two disjoint subsets $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$. Then the $\mathcal{D}_{train}$ dataset (the training set) is used to train the model and the $\mathcal{D}_{val}$ dataset (the validation dataset) is used to test the newly trained model to ensure that a similar performance is observed as on the training set. This way, one can easily monitor both the training and validation errors simultaneously while training the model to identify the ``optimal'' stopping point for training. Most often, the training set will be chosen to represent about 80% of the overall data and the validation will be chose to represent the remaining 20%. This is referred to an 80-20 split of the data. However, depending on the amount of the data and the desired goals, some will utilize 90-10 splits, 75-25 splits, or even 50-50 splits.

- It should be noted that the training set is (intuitively) typically much larger than the validation set.

Visualization of a Train-Test Split



Typical Illustration of Overfitting



K-Fold Cross Validation

Although using a single train-test split of the data is essential to properly evaluate a model's performance, a better method to apply is to use multiple different train-test splits. This is the idea behind K-fold cross validation (CV).

K-Fold Cross Validation

Classical model validation typically consists of a **single train-test split** of the data, using the training set to train the model, and then validating the model with the testing set.

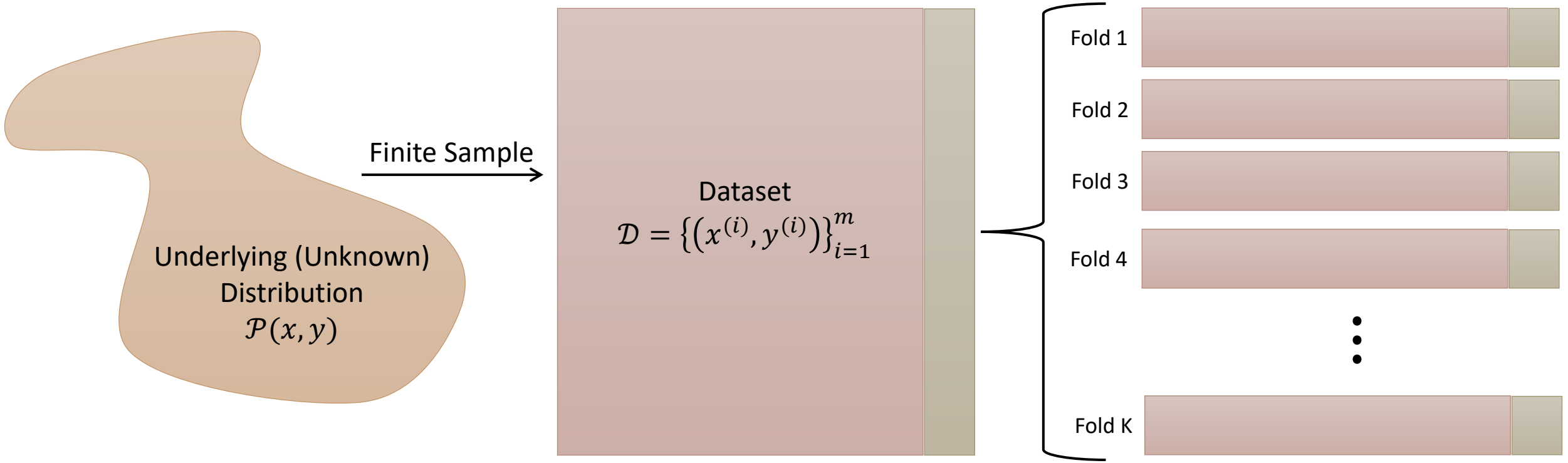
- This is also known as using a **single fold** or a **1-fold CV**.

A superior validation method is to use **K folds**. As opposed to using a single split of the data, K-fold CV uses K different train-test splits of the data, each of which is used to train and validate the model on a different portion of the original data.

Specifically, the original dataset is be split into K **disjoint** subsets. Then, one of the K subsets will be chosen as the testing set for that fold, where all the other subsets are used as training. After the performance of the model has been evaluated on that split, a different subset will be chosen for a new testing set while the others are again used for training. This process is then repeated for a total of K times so that every subset is eventually used as the testing set.

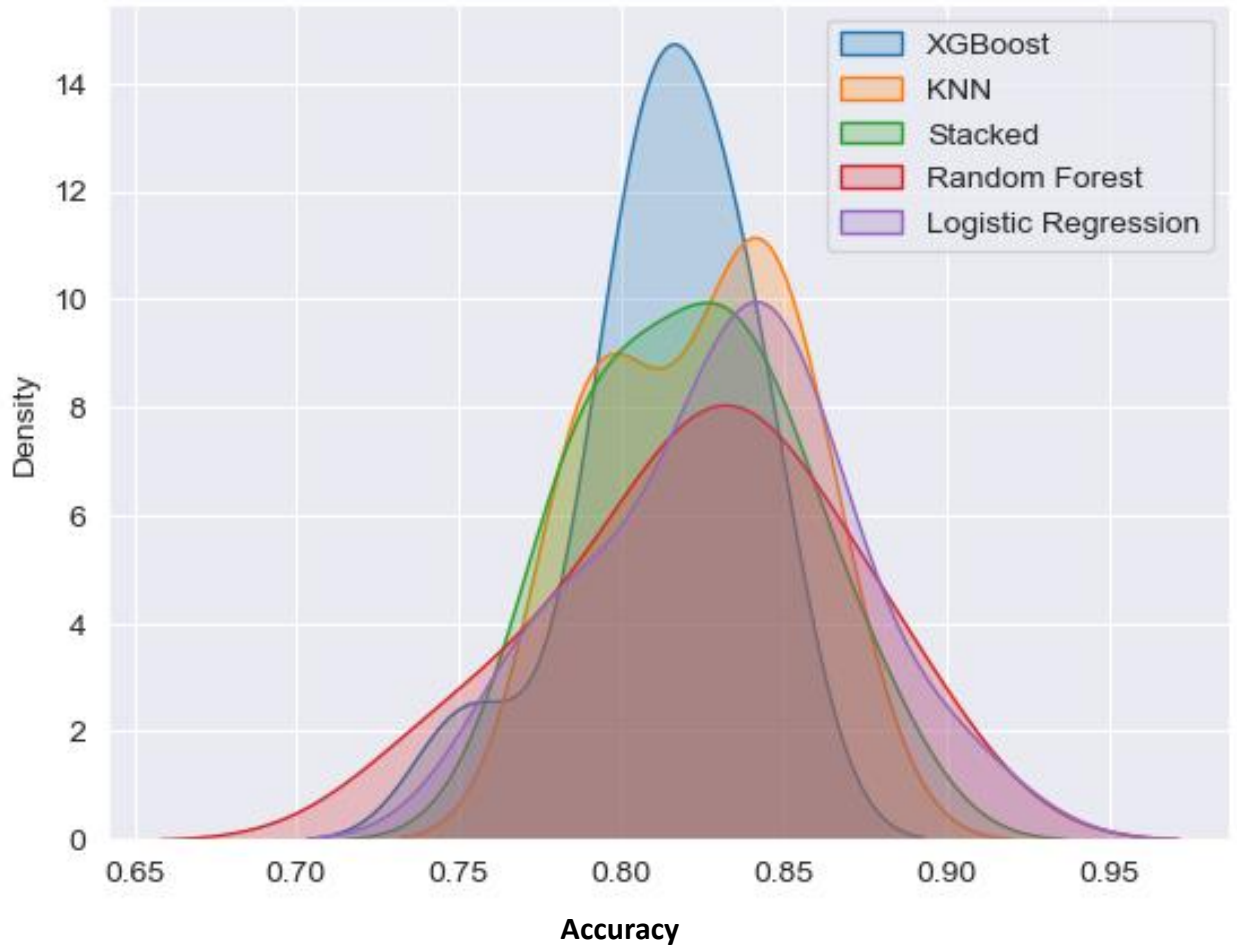
- In this way, all of the data will be utilized as "unseen data" to validate the model on.

Visualization of a K-Fold Validation



Using K-Fold CV to Evaluate Model Performance Distributions

- K-Fold CV is very useful for comparing the performance of different models in a **statistically sound** way.
- When choosing a large enough K, one will obtain enough samples to obtain a representative **distribution of the performance** of a model.
- This model performance distribution will capture a more **accurate** depiction of the real performance of a model when seeing new data.
- This will include the "Expected" (average) performance of a model as well as the "spread" (variance) of outcomes.
- When given several different ML models, one can perform statistical tests to identify if there is a statistically significant difference between two model performances.



Regularization

(Optimization Formulation)

Regularization is another technique that aims at preventing overfitting by employing a penalty term to the objective function being minimized which penalizes more complex models.

Regularization

In optimization, given the function $f: \mathbb{R}^n \to \mathbb{R}$ that one is trying to minimize along with some **regularization function** $r: \mathbb{R}^n \to \mathbb{R}$ and a bound $b \in \mathbb{R}$, this can be formulated as the new constrained problem

$$\min_{x \in \mathbb{R}^n} f(x)$$
$$s.t. \quad r(x) \leq b.$$

This can then be re-written as minimizing the corresponding **Lagrangian function**

$$\min_{x \in \mathbb{R}^n} \mathcal{L}(x; \lambda) = f(x) + \lambda(r(x) - b)\,,$$

Where $\lambda \in \mathbb{R}$ is called a **Lagrange multiplier** and essentially acts as a parameter that enforces the ``amount of weight'' that we give to ensuring that the constraint is satisfied.

- For example, if we want to ensure that the regularization constraint is always satisfied, we will choose λ to be large, as it will make that added term in the objective function dominate the overall value.

It should be mentioned that since we are minimizing this problem over the variables x, we will typically just write this new regularization problem as minimizing the problem

$$\min_{x \in \mathbb{R}^n} f(x) + \lambda r(x)\,.$$

ℓ_1 and ℓ_2 Regularization

The most common choice of regularization functions are some type (or mix) of p-norm, i.e., $r(x) = \|x\|_p$ for some $p \in \{1,2,\dots,\infty\}$.

ℓ_1 Regularization

This form of regularization utilizes the ℓ_1-norm as the regularization function and is written as

$$\min_{x\in\mathbb{R}^n} f(x) + \lambda\|x\|_1 .$$

When $f(x)$ is the least squares loss function of $f(x) = \frac{1}{2m}\|X\theta - y\|_2^2$, then this type of linear regression is referred to as ``**Lasso Regression**''.

- The ℓ_1 regularization induces **sparsity** in its solution by penalizing some weights to be absolutely 0.
- Useful in creating ``simpler'' more interpretable models.

ℓ_2 Regularization

This form of regularization utilizes the ℓ_2-norm as the regularization function and is typically written as

$$\min_{x\in\mathbb{R}^n} f(x) + \frac{1}{2}\lambda\|x\|_2^2 .$$

When $f(x)$ is the least squares loss function of $f(x) = \frac{1}{2m}\|X\theta - y\|_2^2$, then this type of linear regression is referred to as ``**Ridge Regression**''.

- Discourages extreme parameter values without forcing them to take on values of 0.
- Leads to a more evenly distributed impact among features.

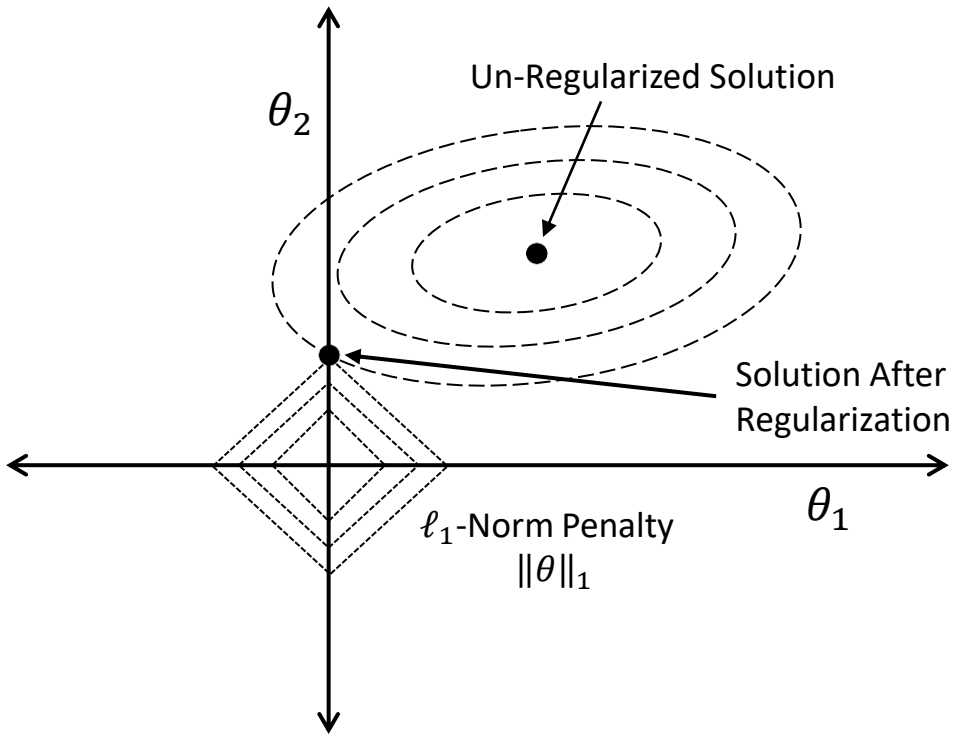
Elastic Net Regression

This is a model that performs linear regression in a least squares fashion, only it implements a regularization function that is a mixture of both the ℓ_1-norm and the ℓ_2-norm as follows:

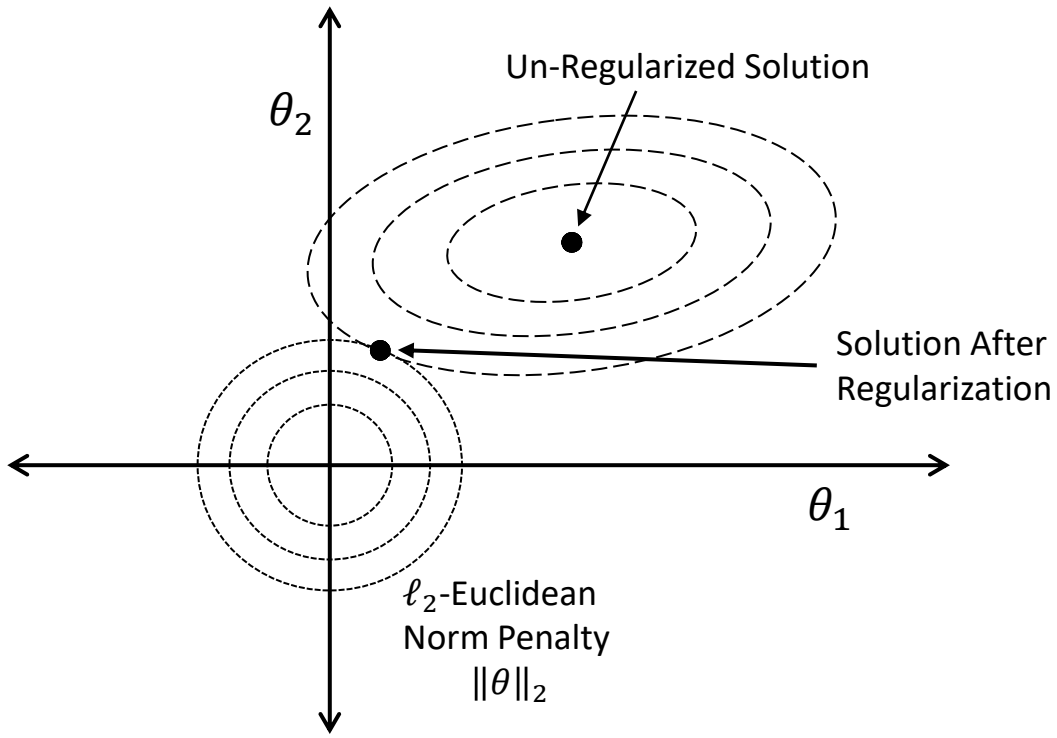
$$\min_{x\in\mathbb{R}^n} f(x) + \lambda\|x\|_1 \frac{1}{2}\lambda\|x\|_2^2 .$$

Visualization of Regularization Techniques

ℓ_1 Regularization



ℓ_2 Regularization

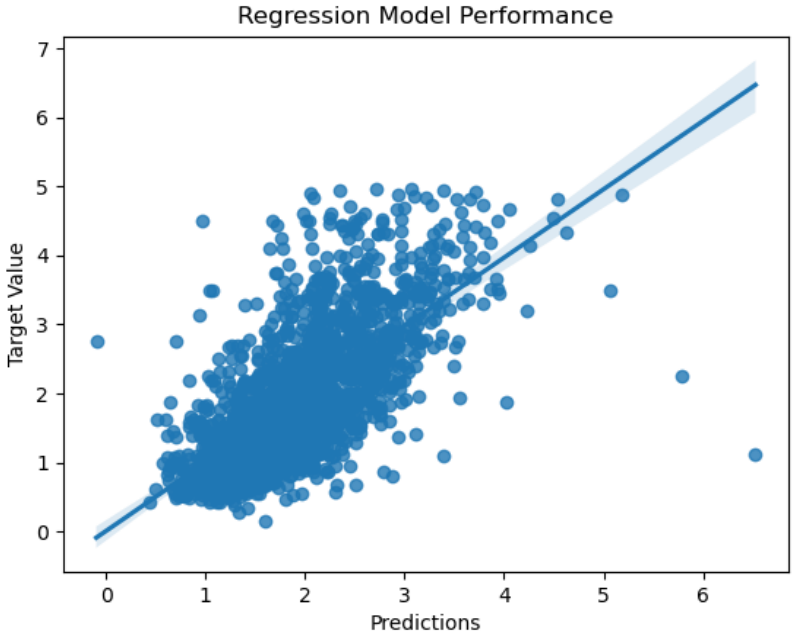


Evaluation Metrics for Regression Models

When it comes to evaluating the performance of a regression model, the two best ways involve reporting two types of numbers, with an associated plot that can be generated for each evaluation metric: these ways are (1) evaluating the **r^2 value** and the other is (2) evaluating the **loss function J** directly.

r^2-Score Metric

One of the best ways to determine your model's accuracy is to plot your model's predictions ($\hat{y}^{(i)}$) against the corresponding target values ($y^{(i)}$). One can then compute the $r^2 \in [0,1]$ (the coefficient of determination), where r is the Pearson correlation coefficient between $\hat{y}^{(i)}$ and $y^{(i)}$. The r^2 metric can be interpreted as the amount of variance in the dataset that is explained by the model.



An $r^2 = 1$ score implies that your model perfectly predicts the target variable, whereas a $r^2 = 0$ score implies that you model has no predictive power.

For the same reasons already mentioned, it is best to evaluate the model on a validation set.

Loss Function Evaluation Over Training

Another way to evaluate your model's performance is to simply evaluate the objective function J that your model is being trained on. Furter, one of the best ways to hyperparameter-tune your model as well as ensuring overfitting is not occurring is to plot the training and validation loss of you model over the total number of training iterations. In this way, one can identify exactly how your model is performing throughout the training process.

This is also a great way to compare the performance of different algorithms in a way that more wholly capture the performance of a model throughout the entire training process.