Philosophical Fundamentals of Machine Learning

Bruce Rushing

Purdue University

Week 3

Table of Contents

Table of Contents

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

There are four types of data:

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

There are four types of data:

1. Binary values, e.x. spam and not spam.

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

There are four types of data:

1. Binary values, e.x. spam and not spam.
2. Category values, e.x. strings like “Yes”, “No”, “blue”, “car”, and so on.

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

There are four types of data:

1. Binary values, e.x. spam and not spam.
2. Category values, e.x. strings like “Yes”, “No”, “blue”, “car”, and so on.
3. Integer values, e.x. age brackets like 10, 20, 30, and so on.

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

There are four types of data:

1. Binary values, e.x. spam and not spam.
2. Category values, e.x. strings like "Yes", "No", "blue", "car", and so on.
3. Integer values, e.x. age brackets like 10, 20, 30, and so on.
4. Real values, e.x. pixel intensities.

Data Types

Whenever approaching a ML problem, *always* first get a feel for the data by looking at it.

There are four types of data:

1. Binary values, e.x. spam and not spam.
2. Category values, e.x. strings like "Yes", "No", "blue", "car", and so on.
3. Integer values, e.x. age brackets like 10, 20, 30, and so on.
4. Real values, e.x. pixel intensities.

The type of ML algorithm we use will often depend on the representation of the data we have. For example, if we are predicting real-valued data we are doing *regression* instead of *classification*. The type of data will also influence how we process the data, i.e. most ML algorithms cannot natively take category values—so we will have to convert it to a better format.

Storing and Manipulating Data

We typically operate with data in ML as lists of *vectors*. Each data point is called an *observation* or *sample*. For example, the j-th sample in a dataset with n features is stored as the row vector:

$$\mathbf{x}^{(j)} = \begin{bmatrix} x_1^{(j)} & x_2^{(j)} & \dots & x_n^{(j)} \end{bmatrix}$$

Storing and Manipulating Data

We typically operate with data in ML as lists of *vectors*. Each data point is called an *observation* or *sample*. For example, the j-th sample in a dataset with n features is stored as the row vector:

$$\mathbf{x}^{(j)} = \begin{bmatrix} x_1^{(j)} & x_2^{(j)} & \dots & x_n^{(j)} \end{bmatrix}$$

A dataset of m samples, $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$, is stored as a (m, n) matrix, $\mathbf{X} \in S^{m \times n}$, where each row is a sample and each column is feature:

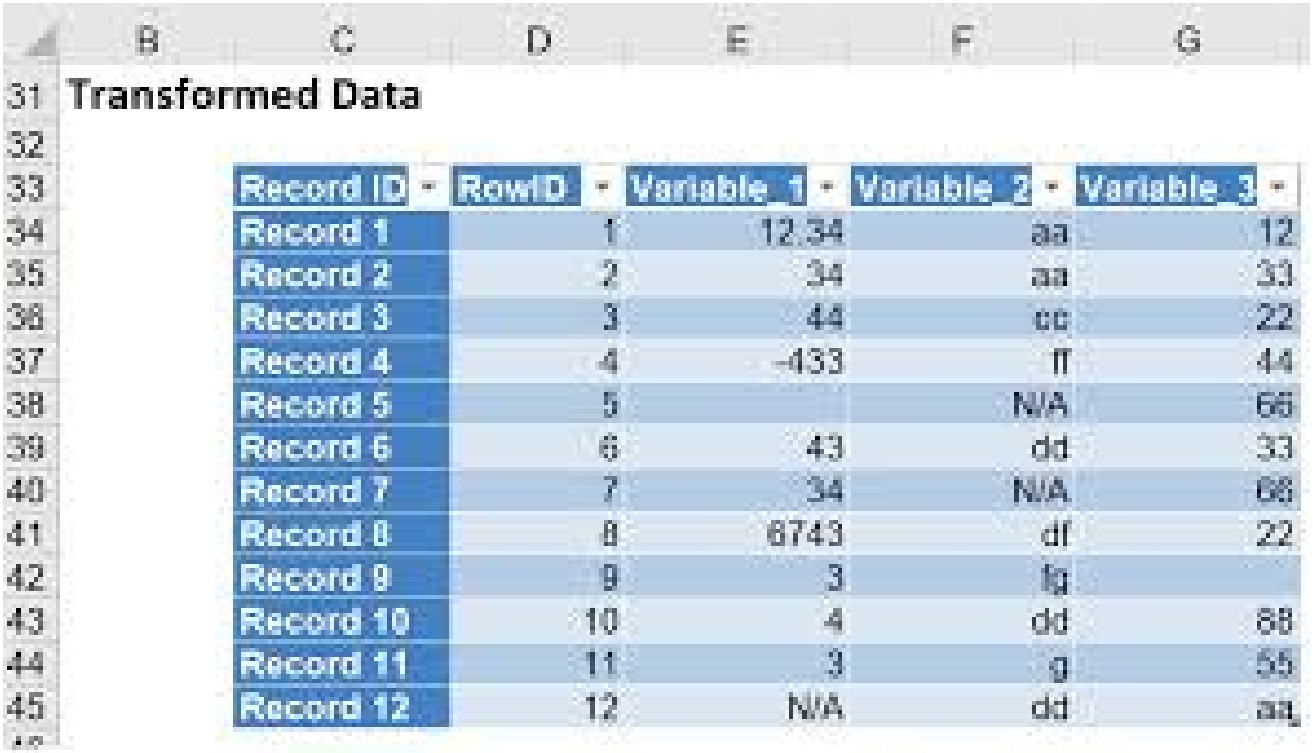
$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{bmatrix}$$

Missing Data

Many datasets are will often have missing values.

Missing Data

Many datasets are will often have missing values.



Transformed Data

Record ID	RowID	Variable_1	Variable_2	Variable_3
Record 1	1	12.34	aa	12
Record 2	2	34	aa	33
Record 3	3	44	cc	22
Record 4	4	-433	ff	44
Record 5	5		N/A	66
Record 6	6	43	dd	33
Record 7	7	34	N/A	66
Record 8	8	6743	df	22
Record 9	9	3	fg	
Record 10	10	4	dd	88
Record 11	11	3	g	55
Record 12	12	N/A	dd	aa

Missing Data

Many datasets are will often have missing values.

Transformed Data

Record ID	RowID	Variable_1	Variable_2	Variable_3
Record 1	1	12.34	aa	12
Record 2	2	34	aa	33
Record 3	3	44	cc	22
Record 4	4	-433	ff	44
Record 5	5		N/A	66
Record 6	6	43	dd	33
Record 7	7	34	N/A	66
Record 8	8	6743	df	22
Record 9	9	3	fg	
Record 10	10	4	dd	88
Record 11	11	3	g	55
Record 12	12	N/A	dd	aa

In these circumstances we can do one of three things:

Missing Data

Many datasets are will often have missing values.

Transformed Data

Record ID	RowID	Variable_1	Variable_2	Variable_3
Record 1	1	12.34	aa	12
Record 2	2	34	aa	33
Record 3	3	44	cc	22
Record 4	4	-433	ff	44
Record 5	5		N/A	66
Record 6	6	43	dd	33
Record 7	7	34	N/A	66
Record 8	8	6743	df	22
Record 9	9	3	fg	
Record 10	10	4	dd	88
Record 11	11	3	g	55
Record 12	12	N/A	dd	aa

In these circumstances we can do one of three things:

1. Drop the sample.

Missing Data

Many datasets are will often have missing values.

Transformed Data

Record ID	RowID	Variable_1	Variable_2	Variable_3
Record 1	1	12.34	aa	12
Record 2	2	34	aa	33
Record 3	3	44	cc	22
Record 4	4	-433	ff	44
Record 5	5		N/A	66
Record 6	6	43	dd	33
Record 7	7	34	N/A	66
Record 8	8	6743	df	22
Record 9	9	3	fg	
Record 10	10	4	dd	88
Record 11	11	3	g	55
Record 12	12	N/A	dd	aa

In these circumstances we can do one of three things:

1. Drop the sample.
2. Drop the feature.

Missing Data

Many datasets are will often have missing values.

Transformed Data

Record ID	RowID	Variable_1	Variable_2	Variable_3
Record 1	1	12.34	aa	12
Record 2	2	34	aa	33
Record 3	3	44	cc	22
Record 4	4	-433	ff	44
Record 5	5		N/A	66
Record 6	6	43	dd	33
Record 7	7	34	N/A	66
Record 8	8	6743	df	22
Record 9	9	3	fg	
Record 10	10	4	dd	88
Record 11	11	3	g	55
Record 12	12	N/A	dd	aa

In these circumstances we can do one of three things:

1. Drop the sample.
2. Drop the feature.
3. Fill in with a dummy.

Data Statistics and Visualization

A key thing to figure out before running any ML algorithms is the *distribution* of the data and to look for any outliers.

Data Statistics and Visualization

A key thing to figure out before running any ML algorithms is the *distribution* of the data and to look for any outliers. We do this by three general methods:

Data Statistics and Visualization

A key thing to figure out before running any ML algorithms is the *distribution* of the data and to look for any outliers. We do this by three general methods:

1. Computing summary statistics.

Data Statistics and Visualization

A key thing to figure out before running any ML algorithms is the *distribution* of the data and to look for any outliers. We do this by three general methods:

1. Computing summary statistics.
2. Visualizing the features with scatter plots.

Data Statistics and Visualization

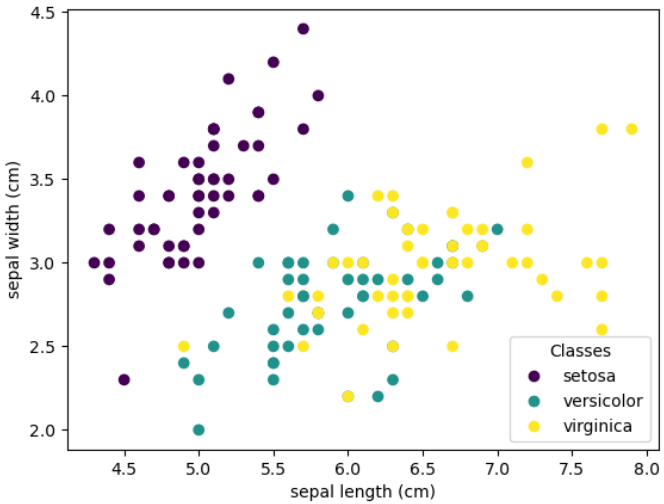
A key thing to figure out before running any ML algorithms is the *distribution* of the data and to look for any outliers. We do this by three general methods:

1. Computing summary statistics.
2. Visualizing the features with scatter plots.
3. Visualizing the distribution with histograms.

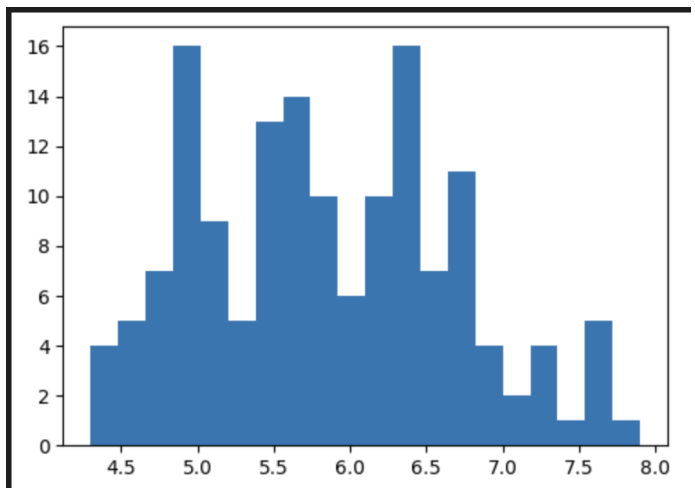
Data Statistics and Visualization

A key thing to figure out before running any ML algorithms is the *distribution* of the data and to look for any outliers. We do this by three general methods:

1. Computing summary statistics.
2. Visualizing the features with scatter plots.
3. Visualizing the distribution with histograms.



(a) Scatter Plot



(b) Histogram

Figure: Examples of scatter plots and histograms from Iris dataset.

Table of Contents

Feature Rescaling

Many ML algorithms are sensitive to extreme values found in the data. For example, neural networks can have trouble learning data with large positive or negative values—even when the data tends to not have outliers.

Feature Rescaling

Many ML algorithms are sensitive to extreme values found in the data. For example, neural networks can have trouble learning data with large positive or negative values—even when the data tends to not have outliers.

A common practice then is to rescale features so that they all assume a value between 0 and 1. While this is not always necessary or desirable (such as in linear regressions), it is generally a good rule of thumb when using many popular algorithms for classification to rescale the features.

Feature Rescaling

Many ML algorithms are sensitive to extreme values found in the data. For example, neural networks can have trouble learning data with large positive or negative values—even when the data tends to not have outliers.

A common practice then is to rescale features so that they all assume a value between 0 and 1. While this is not always necessary or desirable (such as in linear regressions), it is generally a good rule of thumb when using many popular algorithms for classification to rescale the features.

An example rescaling strategy is the *min-max* rescaler, where we rescale each feature x_i as follows:

Feature Rescaling

Many ML algorithms are sensitive to extreme values found in the data. For example, neural networks can have trouble learning data with large positive or negative values—even when the data tends to not have outliers.

A common practice then is to rescale features so that they all assume a value between 0 and 1. While this is not always necessary or desirable (such as in linear regressions), it is generally a good rule of thumb when using many popular algorithms for classification to rescale the features.

An example rescaling strategy is the *min-max* rescaler, where we rescale each feature x_i as follows:

$$x_i^* = \frac{x_i - \min x_i}{\max x_i - \min x_i}$$

Normalization

Rescaling transforms the data features to make them more manageable for certain algorithms without changing the *distribution* of values in those features. In contrast, *normalization* changes the distribution of feature values so that the features are normally distributed.

Normalization

Rescaling transforms the data features to make them more manageable for certain algorithms without changing the *distribution* of values in those features. In contrast, *normalization* changes the distribution of feature values so that the features are normally distributed.

This can be really useful if the data is high variance. For example, if half of the data values cluster around 0 but the other half cluster around 1000, our ML models will tend to ignore the former and focus on the latter when learning and making predictions. We want our ML models to learn patterns in data instead of ignoring certain data.

Normalization

Rescaling transforms the data features to make them more manageable for certain algorithms without changing the *distribution* of values in those features. In contrast, *normalization* changes the distribution of feature values so that the features are normally distributed.

This can be really useful if the data is high variance. For example, if half of the data values cluster around 0 but the other half cluster around 1000, our ML models will tend to ignore the former and focus on the latter when learning and making predictions. We want our ML models to learn patterns in data instead of ignoring certain data.

Note that rescaling doesn't normally fix this: if we rescaled the prior example, we would still have either our values be 0 or 1, and so our model would largely ignore the former still.

One-Hot Vectors

When working with categorical data, we have to encode it into a format that our ML models can use. While we can encode it in an integer format, this has a bad follow-on effect in that it imparts an *ordering* to the data when there isn't any.

One-Hot Vectors

When working with categorical data, we have to encode it into a format that our ML models can use. While we can encode it in an integer format, this has a bad follow-on effect in that it imparts an *ordering* to the data when there isn't any.

Instead, if we want the various feature values to be independent of one another, we use a *one-hot* or *dummy variable* encoding where each category is encoded as an element in binary vectors. For example, if we have categories "red", "blue", and "green", we can encode it as a binary, 3-length vectors:

One-Hot Vectors

When working with categorical data, we have to encode it into a format that our ML models can use. While we can encode it in an integer format, this has a bad follow-on effect in that it imparts an *ordering* to the data when there isn't any.

Instead, if we want the various feature values to be independent of one another, we use a *one-hot* or *dummy variable* encoding where each category is encoded as an element in binary vectors. For example, if we have categories "red", "blue", and "green", we can encode it as a binary, 3-length vectors:

$$red = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad blue = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad green = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Balancing

In the special case of classification, we also have to worry about the relative numbers of our samples with specific class labels. If those numbers are not roughly equal, we might end up with data that is unbalanced in a way that our model does not learn.

Balancing

In the special case of classification, we also have to worry about the relative numbers of our samples with specific class labels. If those numbers are not roughly equal, we might end up with data that is unbalanced in a way that our model does not learn.

For example, suppose we are predicting the binary variable for survival from a disease. Suppose that in our training data, 99% of our samples have the label survival. Then, our model can still accurately predict the training data with 0.99 accuracy just by always guessing survival—regardless of the features.

Balancing

In the special case of classification, we also have to worry about the relative numbers of our samples with specific class labels. If those numbers are not roughly equal, we might end up with data that is unbalanced in a way that our model does not learn.

For example, suppose we are predicting the binary variable for survival from a disease. Suppose that in our training data, 99% of our samples have the label survival. Then, our model can still accurately predict the training data with 0.99 accuracy just by always guessing survival—regardless of the features.

To prevent this, what we do is make sure that our samples are balanced, e.g. we would have training data where 50% survive and 50% do not survive. One technique is called *random sampling* where we randomly duplicate the minority classes of our training data until we have equal quantities. There are other techniques, such as SMOTE, that synthesize new data for balancing.

Table of Contents

Validation and Test Data

We typically split our dataset into three separate sets:

Validation and Test Data

We typically split our dataset into three separate sets:

1. Training Data: the data we use to train our algorithm on.

Validation and Test Data

We typically split our dataset into three separate sets:

1. Training Data: the data we use to train our algorithm on.
2. Validation Data: the data we use to evaluate our algorithm on for overfitting.

Validation and Test Data

We typically split our dataset into three separate sets:

1. Training Data: the data we use to train our algorithm on.
2. Validation Data: the data we use to evaluate our algorithm on for overfitting.
3. Test Data: the data we use to check for real-world performance.

Validation and Test Data

We typically split our dataset into three separate sets:

1. Training Data: the data we use to train our algorithm on.
2. Validation Data: the data we use to evaluate our algorithm on for overfitting.
3. Test Data: the data we use to check for real-world performance.

The difference between validation and test data is that we use validation data during the training process to select models and hyperparameters for those models; test data is strictly to be used only once.

Validation and Test Data

We typically split our dataset into three separate sets:

1. Training Data: the data we use to train our algorithm on.
2. Validation Data: the data we use to evaluate our algorithm on for overfitting.
3. Test Data: the data we use to check for real-world performance.

The difference between validation and test data is that we use validation data during the training process to select models and hyperparameters for those models; test data is strictly to be used only once.

IMPORTANT: you can retrain to improve performance on validation data, but you should never retrain to improve on test data.

k-Fold Cross Validation

A simple hold-out validation set has a performance gaurantees relative to only one specific sample. A better method is to see how our model performs on multiple hold-out validation sets. This is the idea behind *k-fold cross-validation*.

k-Fold Cross Validation

A simple hold-out validation set has a performance gaurantees relative to only one specific sample. A better method is to see how our model performs on multiple hold-out validation sets. This is the idea behind *k-fold cross-validation*. We partition our training data k validation sets and then train on the complement in our training data, and average the model performance on the k validation splits:

k-Fold Cross Validation

A simple hold-out validation set has a performance gaurantees relative to only one specific sample. A better method is to see how our model performs on multiple hold-out validation sets. This is the idea behind *k-fold cross-validation*. We partition our training data k validation sets and then train on the complement in our training data, and average the model performance on the k validation splits:

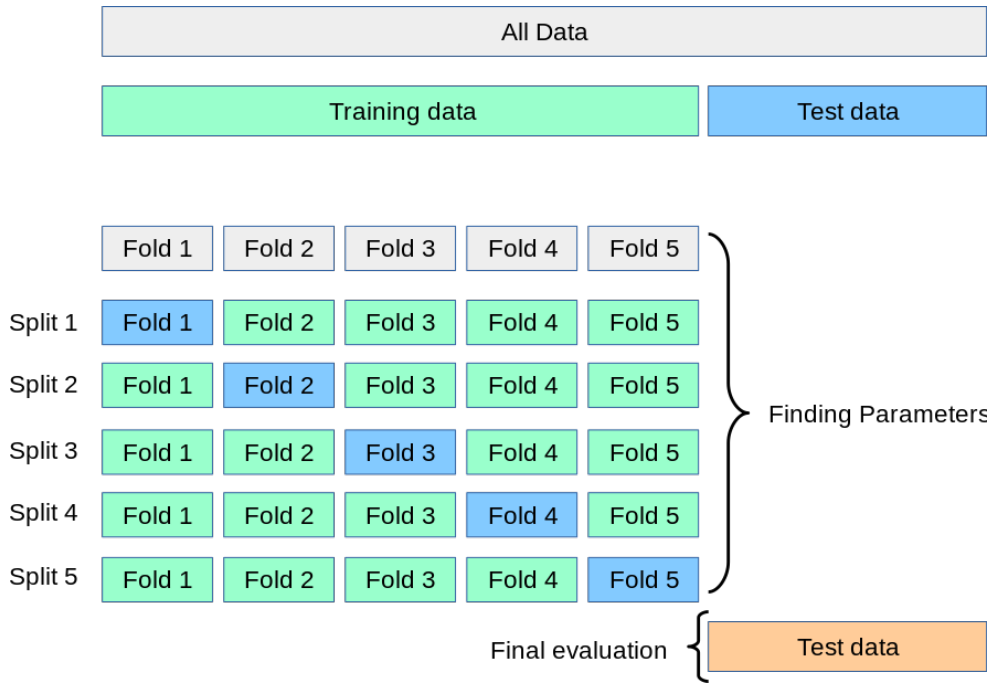


Table of Contents

Maximum Likelihood Estimation

Definition (Likelihood Function)

Suppose we have some data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{m})\}$ that we are trying to learn with a model characterized by parameters $\theta \in \Theta$ that outputs a probabilistic prediction Pr. Then the *likelihood* for that data is defined as:

Maximum Likelihood Estimation

Definition (Likelihood Function)

Suppose we have some data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{m})\}$ that we are trying to learn with a model characterized by parameters $\theta \in \Theta$ that outputs a probabilistic prediction Pr. Then the *likelihood* for that data is defined as:

$$\mathcal{L}_{\mathcal{D}}(\theta) = \Pr(y^{(1)}, \ldots, y^{(m)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, \theta)$$

If we assume that the data is i.i.d., then the likelihood becomes:

Maximum Likelihood Estimation

Definition (Likelihood Function)

Suppose we have some data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{m})\}$ that we are trying to learn with a model characterized by parameters $\theta \in \Theta$ that outputs a probabilistic prediction Pr. Then the *likelihood* for that data is defined as:

$$\mathcal{L}_{\mathcal{D}}(\theta) = \Pr(y^{(1)}, \ldots, y^{(m)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, \theta)$$

If we assume that the data is i.i.d., then the likelihood becomes:

$$\mathcal{L}_{\mathcal{D}}(\theta) = \prod_{i=1}^{m} \Pr(y^{(i)} | \mathbf{x}^{(i)}, \theta)$$

Maximum Likelihood Estimation

Definition (Likelihood Function)

Suppose we have some data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{m})\}$ that we are trying to learn with a model characterized by parameters $\theta \in \Theta$ that outputs a probabilistic prediction Pr. Then the *likelihood* for that data is defined as:

$$\mathcal{L}_{\mathcal{D}}(\theta) = \Pr(y^{(1)}, \ldots, y^{(m)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, \theta)$$

If we assume that the data is i.i.d., then the likelihood becomes:

$$\mathcal{L}_{\mathcal{D}}(\theta) = \prod_{i=1}^{m} \Pr(y^{(i)} | \mathbf{x}^{(i)}, \theta)$$

Example

If the model is a Bernoulli distribution, then the likelihood is just the product of those Bernoullis where $\Pr(y = k | \mathbf{x}, \theta) = p^k (1-p)^{1-k}$,.

Maximum Likelihood Estimation

Maximum Likelihood Estimation

The method of maximum likelihood is to find the model parameters $\theta \in \Theta$ that makes the data most likely:

Maximum Likelihood Estimation

Maximum Likelihood Estimation

The method of maximum likelihood is to find the model parameters $\theta \in \Theta$ that makes the data most likely:

$$\mathcal{L}_{\mathcal{D}}^{\star} = \arg\max_{\theta \in \Theta} \mathcal{L}_{\mathcal{D}}(\theta)$$

Maximum Likelihood Estimation

Maximum Likelihood Estimation

The method of maximum likelihood is to find the model parameters $\theta \in \Theta$ that makes the data most likely:

$$\mathcal{L}_{\mathcal{D}}^{\star} = \arg\max_{\theta \in \Theta} \mathcal{L}_{\mathcal{D}}(\theta)$$

Remark

Maximum likelihood estimation has a Bayesian interpretation. Suppose we wanted to estimate the posterior probability of $\Pr(\theta|\mathcal{D})$, and suppose we wanted the most likely model θ. This is often called the *mode* or *maximum a posteriori* (MAP) estimate. Then the maximum likelihood estimate is just the MAP in cases where we assume a *uniform* prior over possible models.

Negative Log-likelihood

Remark

Because of floating point precision problems when operating on computers, we cannot really use probabilities to estimate likelihoods. Instead, we use logarithm probabilities, which leads to a twist on our optimization problem.

Negative Log-likelihood

Remark

Because of floating point precision problems when operating on computers, we cannot really use probabilities to estimate likelihoods. Instead, we use logarithm probabilities, which leads to a twist on our optimization problem.

Negative Log-likelihood

If we take the negative logarithm of the likelihood function, we end up with the negative log-likelihood:

Negative Log-likelihood

Remark

Because of floating point precision problems when operating on computers, we cannot really use probabilities to estimate likelihoods. Instead, we use logarithm probabilities, which leads to a twist on our optimization problem.

Negative Log-likelihood

If we take the negative logarithm of the likelihood function, we end up with the negative log-likelihood:

$$NLL_{\mathcal{D}}(\theta) = -\sum_{i=1}^{m} \log \Pr(y^{(i)}|\mathbf{x}^{(i)}, \theta)$$

Negative Log-likelihood

Remark

Because of floating point precision problems when operating on computers, we cannot really use probabilities to estimate likelihoods. Instead, we use logarithm probabilities, which leads to a twist on our optimization problem.

Negative Log-likelihood

If we take the negative logarithm of the likelihood function, we end up with the negative log-likelihood:

$$NLL_{\mathcal{D}}(\theta) = -\sum_{i=1}^{m} \log \Pr(y^{(i)}|\mathbf{x}^{(i)}, \theta)$$

Minimizing *NLL* is equivalent to maximizing the likelihood function:

Negative Log-likelihood

Remark

Because of floating point precision problems when operating on computers, we cannot really use probabilities to estimate likelihoods. Instead, we use logarithm probabilities, which leads to a twist on our optimization problem.

Negative Log-likelihood

If we take the negative logarithm of the likelihood function, we end up with the negative log-likelihood:

$$NLL_{\mathcal{D}}(\theta) = -\sum_{i=1}^{m} \log \Pr(y^{(i)}|\mathbf{x}^{(i)}, \theta)$$

Minimizing *NLL* is equivalent to maximizing the likelihood function:

$$\mathcal{L}^{\star}_{\mathcal{D}} = \arg\min_{\theta \in \Theta} NLL_{\mathcal{D}}(\theta)$$

Table of Contents

Linear Regression

In regression, our targets are continuously valued and infinite instead of discrete and finite. The simplest form of regression is to find a *linear transformation* $\Phi : \mathbb{R}^n \to \mathbb{R}$ that best predicts the targets.

Linear Regression

In regression, our targets are continuously valued and infinite instead of discrete and finite. The simplest form of regression is to find a *linear transformation* $\Phi : \mathbb{R}^n \to \mathbb{R}$ that best predicts the targets.

Linear Regressian

Let $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$ and let σ be known, then the linear regression model $f(\mathbf{x}; \theta)$ is defined as:

Linear Regression

In regression, our targets are continuously valued and infinite instead of discrete and finite. The simplest form of regression is to find a *linear transformation* $\Phi : \mathbb{R}^n \to \mathbb{R}$ that best predicts the targets.

Linear Regressian

Let $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$ and let σ be known, then the linear regression model $f(\mathbf{x}; \theta)$ is defined as:

$$f(\mathbf{x}; \theta) = \mathcal{N}(y | \mathbf{x}^\intercal \theta, \sigma^2)$$

Our probabilistic likelihood of a normal distribution with known variance, whose mean is the inner product of input features and weights, θ.

Linear Regression

In regression, our targets are continuously valued and infinite instead of discrete and finite. The simplest form of regression is to find a *linear transformation* $\Phi : \mathbb{R}^n \to \mathbb{R}$ that best predicts the targets.

Linear Regressian

Let $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$ and let σ be known, then the linear regression model $f(\mathbf{x}; \theta)$ is defined as:

$$f(\mathbf{x}; \theta) = \mathcal{N}(y | \mathbf{x}^\intercal \theta, \sigma^2)$$

Our probabilistic likelihood of a normal distribution with known variance, whose mean is the inner product of input features and weights, θ.

Remark

Typically when making predictions, we take the expected value of our model, i.e $\mathbb{E}[f(\mathbf{x}; \theta)]$. Here with our linear regression, this will just be the mean of our normal distribution: $\mathbf{x}^\intercal \theta$.

Ordinary Least Squares

Mean Squared Error

When we make the assumption that our data is i.i.d. and the variance is known, we can analytically solve for the linear regression that maximizes the likelihood. We do this by finding the model that minimizes the negative log-likelihood. Let $\mathbf{X} \in \mathbb{R}^{m \times n}$ be the matrix of our data features and $\mathbf{y}$ the vector of targets from $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$.

Ordinary Least Squares

Mean Squared Error

When we make the assumption that our data is i.i.d. and the variance is known, we can analytically solve for the linear regression that maximizes the likelihood. We do this by finding the model that minimizes the negative log-likelihood. Let $\mathbf{X} \in \mathbb{R}^{m \times n}$ be the matrix of our data features and $\mathbf{y}$ the vector of targets from $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$. Then, throwing away constants, the *NLL* is:

Ordinary Least Squares

Mean Squared Error

When we make the assumption that our data is i.i.d. and the variance is known, we can analytically solve for the linear regression that maximizes the likelihood. We do this by finding the model that minimizes the negative log-likelihood. Let $\mathbf{X} \in \mathbb{R}^{m \times n}$ be the matrix of our data features and $\mathbf{y}$ the vector of targets from $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$. Then, throwing away constants, the *NLL* is:

$$\begin{aligned} NLL_{\mathcal{D}}^{LR}(\theta) &= \frac{1}{2\sigma^2} \sum_{i=1}^{m} (y^{(i)} - \mathbf{x}^{(i)\top}\theta)^2 \\ &= \frac{1}{2\sigma^2} (\mathbf{y} - \mathbf{X}\theta)^2 \end{aligned}$$

Ordinary Least Squares

Mean Squared Error

When we make the assumption that our data is i.i.d. and the variance is known, we can analytically solve for the linear regression that maximizes the likelihood. We do this by finding the model that minimizes the negative log-likelihood. Let $\mathbf{X} \in \mathbb{R}^{m \times n}$ be the matrix of our data features and $\mathbf{y}$ the vector of targets from $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$. Then, throwing away constants, the *NLL* is:

$$\begin{aligned} NLL^{LR}_{\mathcal{D}}(\theta) &= \frac{1}{2\sigma^2} \sum_{i=1}^{m} (y^{(i)} - \mathbf{x}^{(i)\intercal}\theta)^2 \\ &= \frac{1}{2\sigma^2} (\mathbf{y} - \mathbf{X}\theta)^2 \end{aligned}$$

Taking the average across our m samples, this is just the classic *mean squared error*.

Ordinary Least Squares

Ordinary Least Squares

Minimizing the negative log-likelihood loss gives us the *ordinary least squares* (OLS) solution to linear regression, where we solve for where the first derivative is 0 and we ensure our feature matrix has a special property (positive definite):

Ordinary Least Squares

Ordinary Least Squares

Minimizing the negative log-likelihood loss gives us the *ordinary least squares* (OLS) solution to linear regression, where we solve for where the first derivative is 0 and we ensure our feature matrix has a special property (positive definite):

$$\mathcal{L}_{\mathcal{D}}^{\star} = (\mathbf{X}^{\intercal}\mathbf{X})^{-1}\mathbf{X}^{\intercal}\mathbf{y}$$

With our assumptions, this is a global minima and so the optimal solution to our problem.

Ordinary Least Squares

Ordinary Least Squares

Minimizing the negative log-likelihood loss gives us the *ordinary least squares* (OLS) solution to linear regression, where we solve for where the first derivative is 0 and we ensure our feature matrix has a special property (positive definite):

$$\mathcal{L}_{\mathcal{D}}^{\star} = (\mathbf{X}^{\intercal}\mathbf{X})^{-1}\mathbf{X}^{\intercal}\mathbf{y}$$

With our assumptions, this is a global minima and so the optimal solution to our problem.

Remark

If we cannot ensure the conditions for the OLS solution, we use gradient descent to find the best linear regression.

Table of Contents

Regularization

Often when doing linear regression, we will find that our features happen to be correlated with one another. This can lead to our linear regression *overfitting*. To combat this, we use what is called a *regularizer* to make our model better at fitting to our training data.

Regularization

Often when doing linear regression, we will find that our features happen to be correlated with one another. This can lead to our linear regression *overfitting*. To combat this, we use what is called a *regularizer* to make our model better at fitting to our training data.

Regularization and Priors

Regularization can be understood as changing our prior in MLE from a uniform prior to another prior. That is, we want the MAP for our linear regression. That MAP is given by the numerator in Bayes theorem. By taking the negative logarithm of that numerator we have:

Regularization

Often when doing linear regression, we will find that our features happen to be correlated with one another. This can lead to our linear regression *overfitting*. To combat this, we use what is called a *regularizer* to make our model better at fitting to our training data.

Regularization and Priors

Regularization can be understood as changing our prior in MLE from a uniform prior to another prior. That is, we want the MAP for our linear regression. That MAP is given by the numerator in Bayes theorem. By taking the negative logarithm of that numerator we have:

$$\begin{aligned} \Pr(\theta|\mathcal{D}) &\propto \Pr(\mathcal{D}|\theta)\Pr(\theta) \\ -\log\Pr(\theta|\mathcal{D}) &\propto -\log(\Pr(\mathcal{D}|\theta)\Pr(\theta)) \\ &NLL_{\mathcal{D}}(\theta) - \log\Pr(\theta) \end{aligned}$$

Ridge Regression

Ridge Regression

With a linear regression, our $NLL_{\theta}^{LR}(\theta)$ stays the same, i.e. it is the mean squared error, but our prior is given by the standard normal prior over weights:

Ridge Regression

Ridge Regression

With a linear regression, our $NLL_{\theta}^{LR}(\theta)$ stays the same, i.e. it is the mean squared error, but our prior is given by the standard normal prior over weights:

$$\Pr(\theta) = \mathcal{N}(\theta|0, \alpha^{-1}\mathbf{I})$$

Ridge Regression

Ridge Regression

With a linear regression, our $NLL_\theta^{LR}(\theta)$ stays the same, i.e. it is the mean squared error, but our prior is given by the standard normal prior over weights:

$$\Pr(\theta) = \mathcal{N}(\theta|0, \alpha^{-1}\mathbf{I})$$

Ignoring constants, this provides us the following loss function, which is called l_2 loss:

Ridge Regression

Ridge Regression

With a linear regression, our $NLL_\theta^{LR}(\theta)$ stays the same, i.e. it is the mean squared error, but our prior is given by the standard normal prior over weights:

$$\Pr(\theta) = \mathcal{N}(\theta|0, \alpha^{-1}\mathbf{I})$$

Ignoring constants, this provides us the following loss function, which is called l_2 loss:

$$NLL_{\mathcal{D}}^{RR}(\theta) = \sum_{i=1}^{m}(y^{(i)} - \mathbf{x}^{(i)\intercal}\theta)^2 + \alpha\sum_{i=0}^{n}\theta_i^2$$

Ridge Regression

Ridge Regression

With a linear regression, our $NLL_\theta^{LR}(\theta)$ stays the same, i.e. it is the mean squared error, but our prior is given by the standard normal prior over weights:

$$\Pr(\theta) = \mathcal{N}(\theta|0, \alpha^{-1}\mathbf{I})$$

Ignoring constants, this provides us the following loss function, which is called l_2 loss:

$$NLL_{\mathcal{D}}^{RR}(\theta) = \sum_{i=1}^{m}(y^{(i)} - \mathbf{x}^{(i)\intercal}\theta)^2 + \alpha\sum_{i=0}^{n}\theta_i^2$$

We use this loss function when fitting our regression. The resulting model is called *ridge regression*. The parameter α is the weight penalty we give to large weights θ.

Table of Contents

Logistic Regression

The direct analog to linear regression for classification are linear classifiers. All linear classifiers involve the application of a linear transformation $\Phi : \mathbb{R}^n \to \mathbb{R}^d$ of $n-1$ features and d targets and apply a non-linear function to decide which class to predict.

Logistic Regression

The direct analog to linear regression for classification are linear classifiers. All linear classifiers involve the application of a linear transformation $\Phi : \mathbb{R}^n \to \mathbb{R}^d$ of $n-1$ features and d targets and apply a non-linear function to decide which class to predict.

Definition (Logistic Sigmoid)

The *logistic sigmoid* function $\sigma : \mathbb{R} \to \mathbb{R}$ is given by:

Logistic Regression

The direct analog to linear regression for classification are linear classifiers. All linear classifiers involve the application of a linear transformation $\Phi : \mathbb{R}^n \to \mathbb{R}^d$ of $n-1$ features and d targets and apply a non-linear function to decide which class to predict.

Definition (Logistic Sigmoid)

The *logistic sigmoid* function $\sigma : \mathbb{R} \to \mathbb{R}$ is given by:

$$\sigma(x) := \frac{1}{1 + \exp(-x)}$$

Logistic Regression

The direct analog to linear regression for classification are linear classifiers. All linear classifiers involve the application of a linear transformation $\Phi : \mathbb{R}^n \to \mathbb{R}^d$ of $n-1$ features and d targets and apply a non-linear function to decide which class to predict.

Definition (Logistic Sigmoid)

The *logistic sigmoid* function $\sigma : \mathbb{R} \to \mathbb{R}$ is given by:

$$\sigma(x) := \frac{1}{1 + \exp(-x)}$$

Binary Logistic Regression

Let our targets $Y : \Omega \to \{0, 1\}$, $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$, then the *binary logistic regression* model is:

Logistic Regression

The direct analog to linear regression for classification are linear classifiers. All linear classifiers involve the application of a linear transformation $\Phi : \mathbb{R}^n \to \mathbb{R}^d$ of $n-1$ features and d targets and apply a non-linear function to decide which class to predict.

Definition (Logistic Sigmoid)

The *logistic sigmoid* function $\sigma : \mathbb{R} \to \mathbb{R}$ is given by:

$$\sigma(x) := \frac{1}{1 + \exp(-x)}$$

Binary Logistic Regression

Let our targets $Y : \Omega \to \{0, 1\}$, $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$, then the *binary logistic regression* model is:

$$f(\mathbf{x}; \theta) = \text{Ber}(y | \sigma(\mathbf{x}^\intercal \theta))$$

Logistic Regression

Remark

We can extend the binary logistic regression to a multinomial.

Logistic Regression

Remark

We can extend the binary logistic regression to a multinomial.

Definition (Softmax)

The softmax function $softmax : \mathbb{R}^n \to \mathbb{R}^d$ is:

$$softmax(\mathbf{x})_i := \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

Logistic Regression

Remark

We can extend the binary logistic regression to a multinomial.

Definition (Softmax)

The softmax function $softmax : \mathbb{R}^n \to \mathbb{R}^d$ is:

$$softmax(\mathbf{x})_i := \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

Multinomial Logistic Regression

Let our targets Y be categorical, $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$, then the *multinomial logistic regression* models is:

Logistic Regression

Remark

We can extend the binary logistic regression to a multinomial.

Definition (Softmax)

The softmax function $softmax : \mathbb{R}^n \to \mathbb{R}^d$ is:

$$softmax(\mathbf{x})_i := \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

Multinomial Logistic Regression

Let our targets Y be categorical, $\mathbf{x} \in \mathbb{R}^n$ where $\mathbf{x} = [1, x_1, \ldots, x_{n-1}]^\intercal$. If $\theta = [\theta_0, \theta_1, \ldots, \theta_{n-1}]^\intercal$, then the *multinomial logistic regression* models is:

$$f(\mathbf{x}; \theta) = \text{Cat}(y | softmax(\mathbf{x}^\intercal \theta))$$

Logistic Regression Loss Function

Remark

Just like in linear regression, we can execute MLE by minimizing the negative log-likelihood loss in binary logistic regression.

Logistic Regression Loss Function

Remark

Just like in linear regression, we can execute MLE by minimizing the negative log-likelihood loss in binary logistic regression.

Binary Cross-Entropy

The negative log-likelihood of logistic regression with data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$ under i.i.d. assumptions is given by:

Logistic Regression Loss Function

Remark

Just like in linear regression, we can execute MLE by minimizing the negative log-likelihood loss in binary logistic regression.

Binary Cross-Entropy

The negative log-likelihood of logistic regression with data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$ under i.i.d. assumptions is given by:

$$
\begin{aligned}
NLL_{\mathcal{D}}^{ce} &= -\log \prod_{i=1}^{m} \text{Ber}(y^{(i)} | \sigma(\mathbf{x}^{(i)\intercal}\theta)) \\
&= \sum_{i=1}^{m} -\log(\sigma(\mathbf{x}^{(i)\intercal}\theta)^{y^{(i)}} \cdot (1 - \sigma(\mathbf{x}^{(i)\intercal}\theta))^{1-y^{(i)}}) \\
&= \sum_{i=1}^{m} -[y^{(i)} \log \sigma(\mathbf{x}^{(i)\intercal}\theta) + (1 - y^{(i)}) \log(1 - \sigma(\mathbf{x}^{(i)\intercal}\theta))]
\end{aligned}
$$

Optimizing Logistic Regression

We can extend the binary cross-entropy function to the multinomial case with the general cross-entropy loss. Unfortunately, in both cases, we cannot analytically compute the model parameters that minimize the NLL. So we have to use an optimization scheme, namely **gradient descent**.

Table of Contents

Derivatives

Remark

The key idea we will leverage is the slope or contour of the loss landscape relative to our model parameters to guide us in finding optimal parameters. This is called *hill climbing* or *hill descending*.

Derivatives

Remark

The key idea we will leverage is the slope or contour of the loss landscape relative to our model parameters to guide us in finding optimal parameters. This is called *hill climbing* or *hill descending*.

Definition (Derivative)

The *derivative* of function $f : \mathbb{R} \to \mathbb{R}$ at point $x \in \mathbb{R}$ is:

Derivatives

Remark

The key idea we will leverage is the slope or contour of the loss landscape relative to our model parameters to guide us in finding optimal parameters. This is called *hill climbing* or *hill descending*.

Definition (Derivative)

The *derivative* of function $f : \mathbb{R} \to \mathbb{R}$ at point $x \in \mathbb{R}$ is:

$$\frac{df}{dx} := \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Derivatives

Remark

The key idea we will leverage is the slope or contour of the loss landscape relative to our model parameters to guide us in finding optimal parameters. This is called *hill climbing* or *hill descending*.

Definition (Derivative)

The *derivative* of function $f : \mathbb{R} \to \mathbb{R}$ at point $x \in \mathbb{R}$ is:

$$\frac{df}{dx} := \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Remark

Intuitively, we can think of the derivate, when it exists, as giving us the *tangent* line at a specific point.

Gradients

Definition (Partial Derivative)

For a function $f : \mathbb{R}^n \to \mathbb{R}$ and vector $\mathbf{x} \in \mathbb{R}^n$ with elements $x_1, \ldots, x_n$, the *partial derivatives* of f at $\mathbf{x}$ are:

Gradients

Definition (Partial Derivative)

For a function $f : \mathbb{R}^n \to \mathbb{R}$ and vector $\mathbf{x} \in \mathbb{R}^n$ with elements $x_1, \ldots, x_n$, the *partial derivatives* of f at $\mathbf{x}$ are:

$$\begin{aligned} \frac{\partial f}{\partial x_1} &= \lim_{h \to 0} \frac{f(x_1+h, x_2, \ldots, x_n) - f(\mathbf{x})}{h} \\ &\vdots \\ \frac{\partial f}{\partial x_n} &= \lim_{h \to 0} \frac{f(x_1, \ldots, x_{n-1}, x_n+h) - f(\mathbf{x})}{h} \end{aligned}$$

Gradients

Definition (Partial Derivative)

For a function $f : \mathbb{R}^n \to \mathbb{R}$ and vector $\mathbf{x} \in \mathbb{R}^n$ with elements $x_1, \ldots, x_n$, the *partial derivatives* of f at $\mathbf{x}$ are:

$$\begin{aligned} \frac{\partial f}{\partial x_1} &= \lim_{h \to 0} \frac{f(x_1+h, x_2, \ldots, x_n) - f(\mathbf{x})}{h} \\ &\vdots \\ \frac{\partial f}{\partial x_n} &= \lim_{h \to 0} \frac{f(x_1, \ldots, x_{n-1}, x_n+h) - f(\mathbf{x})}{h} \end{aligned}$$

Definition (Gradient)

The *gradient* of a function f relative to point $\mathbf{x}$, $\nabla_{\mathbf{x}} f$, is defined as the row vector of partial derivatives of f at $\mathbf{x}$:

Gradients

Definition (Partial Derivative)

For a function $f : \mathbb{R}^n \to \mathbb{R}$ and vector $\mathbf{x} \in \mathbb{R}^n$ with elements $x_1, \ldots, x_n$, the *partial derivatives* of f at $\mathbf{x}$ are:

$$\begin{aligned} \frac{\partial f}{\partial x_1} &= \lim_{h \to 0} \frac{f(x_1+h, x_2, \ldots, x_n) - f(\mathbf{x})}{h} \\ &\vdots \\ \frac{\partial f}{\partial x_n} &= \lim_{h \to 0} \frac{f(x_1, \ldots, x_{n-1}, x_n+h) - f(\mathbf{x})}{h} \end{aligned}$$

Definition (Gradient)

The *gradient* of a function f relative to point $\mathbf{x}$, $\nabla_{\mathbf{x}} f$, is defined as the row vector of partial derivatives of f at $\mathbf{x}$:

$$\nabla_{\mathbf{x}} f = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} & \frac{\partial f(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{1 \times n}$$

Gradient Descent Algorithm

Remark

Our goal is to minimize our loss function $l_{\mathcal{D}}$ with our model $f(\mathbf{x};\theta)$:

$$\min_{\theta \in \Theta} l_{\mathcal{D}}(f(\mathbf{x};\theta))$$

To do that, we essentially find the spot where our gradient is 0 by iteratively following it downhill.

Gradient Descent Algorithm

Remark

Our goal is to minimize our loss function $l_{\mathcal{D}}$ with our model $f(\mathbf{x}; \theta)$:

$$\min_{\theta \in \Theta} l_{\mathcal{D}}(f(\mathbf{x}; \theta))$$

To do that, we essentially find the spot where our gradient is 0 by iteratively following it downhill.

Gradient Descent

Let $l : \mathbb{R}^n \to \mathbb{R}$ be some loss function. The *gradient descent* algorithm is start with some initial values of $\theta \in \mathbb{R}^n$, $\theta^{(0)}$, and to iteratively apply the following rule until reaching some stopping criteria:

Gradient Descent Algorithm

Remark

Our goal is to minimize our loss function $l_{\mathcal{D}}$ with our model $f(\mathbf{x};\theta)$:

$$\min_{\theta\in\Theta} l_{\mathcal{D}}(f(\mathbf{x};\theta))$$

To do that, we essentially find the spot where our gradient is 0 by iteratively following it downhill.

Gradient Descent

Let $l:\mathbb{R}^n\to\mathbb{R}$ be some loss function. The *gradient descent* algorithm is start with some initial values of $\theta\in\mathbb{R}^n$, $\theta^{(0)}$, and to iteratively apply the following rule until reaching some stopping criteria:

$$\theta^{(i+1)} = \theta^{(i)} - \eta_i(\nabla_\theta l_{\mathcal{D}}(f(\mathbf{x};\theta))(\theta^{(i)}))$$

The variable η_i is called the *learning rate* and so long as we choose a small enough learning rate, our algorithm will converge to a local minima.

Gradient Descent