**Chapter 4 - Training Models**

Chapter 4 focuses on understanding how Machine Learning models and their training algorithms work. While you can use them as black boxes, knowing the details helps in selecting the right model and algorithm, choosing hyperparameters, debugging, and error analysis.

The chapter covers the following models and training algorithms:

**Linear Regression**

* A simple model that makes predictions by computing a **weighted sum of input features plus a bias term**.
* A linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (intercept term).  
  ŷ = θ0 + θ1x1 + θ2x2 + ⋯ + θnxn  
  • ŷ is the predicted value.  
  • n is the number of features.  
  • xi is the ith feature value.  
  • θj is the jth model parameter (including the bias term θ0 and the feature weights θ1, θ2, ⋯, θn).
* Training involves setting parameters theta(θ) to best fit the training data, typically by minimizing a cost function.
* The most common performance measure for regression is the Root Mean Square Error (RMSE). Minimizing the Mean Square Error (MSE) is simpler and achieves the same result.
* The MSE cost function for a Linear Regression model is
  + MSE=1/m(​i=1∑m​(y^​(i)−y(i))2)

**Methods to train Linear Regression:**

1. **Normal Equation**: A direct, closed-form mathematical equation to find the theta(θ) that minimizes the cost function.
   * The equation is

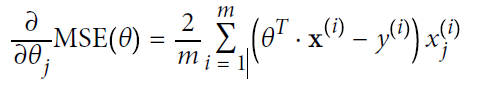


* + $hat{ {theta}}$ is the value of $ {theta}$ that minimizes the cost function.
  + $ {y}$ is the vector of target values.
  + Uses NumPy's np.linalg.inv() for matrix inverse and dot() for matrix multiplication.
  + Scikit-Learn's LinearRegression class uses the Normal Equation (or specifically, the SVD approach which is related). SVD= Singular Value Decomposition.
  + Computational complexity: O(n^2.4) to O(n^3) for matrix inversion, O(n^2) for SVD. Linear with respect to the number of instances O(m). Slow for a large number of features.
  + Pseudoinverse ($\mathbf{X}^{+}$) based on Singular Value Decomposition (SVD) is used by Scikit-Learn and handles cases where $\mathbf{X}^T\mathbf{X}$ is not invertible.

1. **Gradient Descent (GD)**: An iterative optimization algorithm that gradually tweaks parameters to minimize the cost function.
   * Starts with random parameter values (random initialization) and takes steps in the direction of the steepest slope downhill of the cost function.
   * Stops when the gradient is zero (minimum reached).
   * **Learning rate (η)** is a crucial hyperparameter determining step size.
     + Too small: slow convergence.
     + Too high: algorithm may diverge.
   * Can get stuck in local minima for non-convex cost functions.
   * Requires features to have similar scales to converge quickly. Use StandardScaler from Scikit-Learn.
   * Training a model means searching in the model's parameter space. For Linear Regression, the cost function (MSE) is convex, so the minimum is at the bottom of a bowl shape.
   * To implement GD, compute the gradient of the cost function with respect to each parameter. This is the vector of partial derivatives



**Gradient Descent Variants:**

1. **Batch Gradient Descent**: Computes the gradient using the **entire training set** at every step.
   * Cost function partial derivatives:
   * Gradient vector:

 = 

Updated weights:

, calculated above

* + Slow on very large training sets because it uses the whole batch.
  + Scales well with the number of features.
  + Reaches the minimum precisely if the cost function is convex.

1. **Stochastic Gradient Descent (SGD)**: Picks a **random instance** from the training set at each step to compute gradients.
   * Much faster because it manipulates very little data per iteration.
   * Can train on huge training sets that don't fit in memory (out-of-core learning).
   * Randomness helps escape local minima for non-convex cost functions.
   * Randomness also means it never truly settles at the minimum, it walks around it.
   * To converge to the minimum, the learning rate must be gradually reduced using a **learning schedule**.
   * Implemented in Scikit-Learn's SGDRegressor class (for regression).
   * Faster for large numbers of instances (large m).
2. **Mini-batch Gradient Descent**: Computes gradients on **small random sets of instances (mini-batches)**.
   * Performance boost from hardware optimization (e.g., GPUs).
   * Less erratic progress than SGD, walks closer to the minimum.
   * May be harder to escape local minima than SGD.
   * Faster for large m, like SGD.

**Polynomial Regression**

* Used to fit **nonlinear data** using a linear model by adding powers of each feature as new features.
* Example: add $x\_1^2$ as a new feature.
* Implemented in Scikit-Learn using PolynomialFeatures transformer.

**Learning Curves**

* Plots of the model's performance on the **training set and validation set** as a function of training set size (or iteration).
* Useful for diagnosing **underfitting** and **overfitting**.
* **Underfitting**: Both training and validation errors are high and close together, and plateau. Adding more data won't help.
* **Overfitting**: Training error is much lower than validation error, and there is a significant gap between the curves. Validation error may increase after a certain training set size or iteration. Adding more data can help reduce the gap.

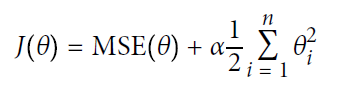
**Bias/Variance Tradeoff**

* A model's generalization error can be decomposed into three errors:
  + **Bias**: Due to wrong assumptions in the model (e.g., assuming linearity when data is quadratic). High bias implies **underfitting**.
  + **Variance**: Due to the model's sensitivity to small variations in the training data. High variance implies **overfitting**.
  + **Irreducible error**: Due to noise in the data itself. Cannot be reduced by model choice; requires data cleaning.
* Increasing model complexity typically increases variance and reduces bias, and vice versa.

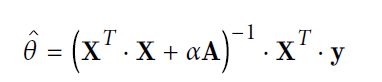
**Regularized Linear Models**

* Techniques to reduce **overfitting** by constraining the model.
* For linear models, this is typically done by constraining the **weights**.
* Regularization term is added to the cost function during training.
* Require scaling the data before use.

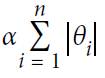
1. **Ridge Regression (L2 regularization)**:
   * Adds to the MSE cost function.

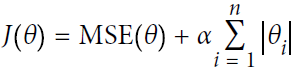


* + Forces weights to be as small as possible.
  + Hyperparameter $\alpha$ controls regularization strength ($\alpha=0$ is Linear Regression). Increasing $\alpha$ leads to flatter predictions, reducing variance but increasing bias.
  + Closed-form solution:



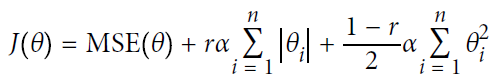
* + Can also be trained using Gradient Descent with an L2 penalty.
  + Implemented in Scikit-Learn as Ridge and SGDRegressor(penalty="l2").
  + A good default method to use, instead of Linear Regression.

1. **Lasso Regression (L1 regularization)**:
   * Adds  to the MSE cost function.



* + Tends to completely eliminate weights of the least important features (sets them to zero). Performs automatic **feature selection**. Outputs a **sparse model**.
  + The cost function is not differentiable at $\theta\_i=0$, but Gradient Descent can still be used with a subgradient vector.
  + Implemented in Scikit-Learn as Lasso and SGDRegressor(penalty="l1").

1. **Elastic Net**: A blend of Ridge and Lasso regularization.



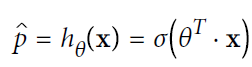
* + Mix ratio $r$ (l1\_ratio in Scikit-Learn) controls the blend. $r=0$ is Ridge, $r=1$ is Lasso.
  + Generally preferred over Lasso because it performs better when features are correlated.
  + Implemented in Scikit-Learn as ElasticNet.

1. **Early Stopping**: Regularization technique for iterative algorithms (like GD) where training is stopped as soon as the **validation error reaches a minimum**.
   * Prevents the model from overfitting after the optimal point is reached.
   * Implemented in Scikit-Learn with warm\_start=True and monitoring the validation error. With warm\_start=True, when the fit() method is called, it just continues

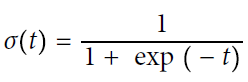
training where it left off instead of restarting from scratch.

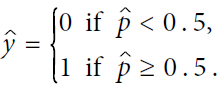
**Logistic Regression(Logit Regression)**

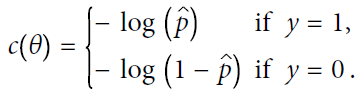
* Commonly used binary classifier to **estimate the probability** an instance belongs to a particular class.
* If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled “1”), or else it predicts that it does not (i.e., it belongs to the negative class, labeled “0”).
* Computes a weighted sum of features plus bias, then outputs the **logistic** (sigmoid) of the result.
* Estimated probability



* The logistic function is



* 
* **Decision boundary** is where $\mathbf{x}^T\boldsymbol{\theta} = 0$, which is linear.
* Training objective is to set $\boldsymbol{\theta}$ to estimate high probabilities for positive instances (y=1) and low for negative (y=0).
* Cost function for a single instance:



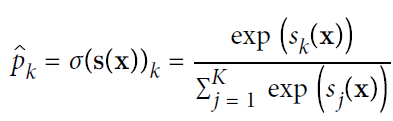
* The cost function for the whole training set is the average over all instances.
* No closed-form solution to minimize the cost function, but it is convex. Gradient Descent (and its variants) can be used.
* Partial derivatives of the cost function are similar to Linear Regression, based on prediction error.
* Implemented in Scikit-Learn as LogisticRegression.
* Can be regularized with L1 or L2 penalties (L2 by default in Scikit-Learn). Regularization strength controlled by hyperparameter C (inverse of $\alpha$). The higher the C, the lower the regularization.

**Softmax Regression**

* A generalization of Logistic Regression to support **multiple classes directly**. Also called Multinomial Logistic Regression.
* Computes a score $s\_k(\mathbf{x})$ for each class, equation similar to Linear Regression.



* Estimates the probability of each class by applying the **softmax function** (normalized exponential) to the scores.
* Softmax function:

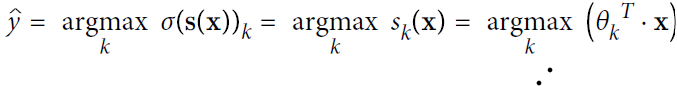


• *K* is the number of classes.

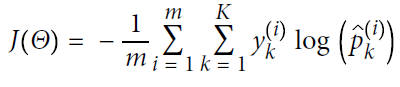
• **s**(**x**) is a vector containing the scores of each class for the instance **x**.

• *σ*(**s**(**x**))*k* is the estimated probability that the instance **x** belongs to class *k* given the scores of each class for that instance.

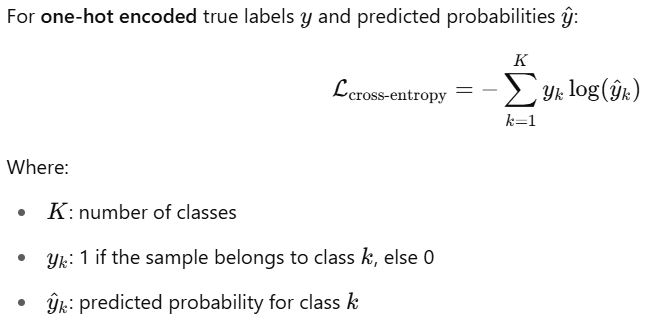
* Predicts the class $k$ with the highest score/probability.
* Only used with **mutually exclusive classes**.
* Training objective is to have high probability for the target class.
* Softmax Classifier target prediction



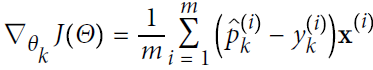
* Minimizes the **cross entropy** cost function because it penalizes the model when it estimates a low probability for target class.



* + $y\_k^{(i)}$ is 1 if instance $i$ belongs to class $k$, 0 otherwise.
  + Equivalent to Logistic Regression cost for K=2.
  + Or simply,



* Gradient Descent can be used to minimize the cross entropy cost.
* Cross entropy gradient vector for class k:



* Implemented in Scikit-Learn's LogisticRegression by setting multi\_class to "multinomial" and using a supporting solver (e.g., "lbfgs"). Applies L2 regularization by default.