Machine Learning Fundamentals and SageMaker Model Development

Regression

Linear Regression: A linear regression model predicts a continuous target variable as a linear combination of input features. It assumes a straight-line (affine) relationship $y = w^T x + b$ between features and output, and finds the parameters w, b that minimize prediction error (e.g. using least squares). For example, one might predict a student's exam score based on hours studied: here hours is the input x and score is the output y (continuous). Linear regression is simple, interpretable and often a good starting model; many more complex methods build on its concepts.

Logistic Regression: Unlike linear regression, logistic regression is used for classification (often binary). It models the log-odds of a binary outcome as a linear function of inputs. Concretely, it computes $z = w^T x + b$ and applies a sigmoid (logistic) function $\sigma(z) = 1/(1 + e^{-z})$ to output a probability between 0 and 1 of belonging to class 1. The model is trained by maximizing the likelihood (minimizing cross-entropy) on labeled data. For example, logistic regression can predict whether an email is spam (class 1) or not spam (class 0) based on text features. It learns a decision boundary (a linear hyperplane) but outputs probabilistic scores, which we threshold (e.g. at 0.5) to assign a class. Unlike linear regression's continuous output, logistic regression explicitly models the probability of class membership using the sigmoid function.

Both linear and logistic regression are **supervised** learning methods: they require labeled examples (x, y) . SageMaker's built-in **Linear Learner** implements these models efficiently. It learns either a real-valued function $w^T x$ for regression or a linear threshold for classification, automatically handling binary and multiclass cases.

Classification

Decision Trees: A decision tree is a non-parametric supervised algorithm for classification (and regression). It creates a tree of decisions based on feature tests, partitioning the feature space into regions with (ideally) homogeneous labels. Each internal node tests a feature (e.g. "Is age > 30?"), splitting the data, and leaf nodes assign a class label. This resembles a flowchart of if-then rules. Trees use measures like information gain or Gini impurity to choose splits that best separate classes. Because of their hierarchical structure, decision trees are easy to interpret. However, very deep trees can overfit (learn noise) unless pruned. In practice, one often controls tree depth to avoid overly complex trees.

Support Vector Machines (SVM): An SVM is a powerful supervised classifier. It finds the optimal separating hyperplane between classes by maximizing the margin (distance) between the nearest points of each class. In 2D this is a line, in higher dimensions a hyperplane. The "support vectors" are the training points closest to this boundary. Maximizing the margin makes the classifier robust to new data. For linearly inseparable data, SVMs use kernel functions (e.g. RBF/Gaussian) to implicitly map inputs into a higher-dimensional space where a linear separator

can be found. SVMs can thus handle nonlinear boundaries via the kernel trick, but even without kernels, a linear SVM is often effective for well-separated data.

K-Nearest Neighbors (KNN): KNN is a simple instance-based (lazy) classifier. It predicts a new point's class by looking at its K nearest neighbors in the training set (using a distance metric like Euclidean). For classification, it takes a majority vote of those neighbors' labels; for regression, it might average their values. For example, to classify a new flower, one finds the K closest flowers in feature space and chooses the most common species among them. KNN makes no training except storing the data; all computation happens at inference. It works best when similar points truly have the same label. A key parameter is K: small K can overfit (high variance), large K can underfit (high bias).

Bias-Variance Tradeoff

Every predictive model has two primary sources of error: **bias** and **variance**. Bias is the error from oversimplified assumptions (underfitting), while variance is the error from sensitivity to small fluctuations in the training set (overfitting). A high-bias model (e.g. a too-simple linear model on nonlinear data) consistently misses the true patterns and makes large errors on both training and new data. A high-variance model (e.g. a very deep tree or high-degree polynomial) fits the training data very closely (possibly even noise), but changes drastically with different training samples, leading to poor generalization. The bias-variance tradeoff states that reducing one often increases the other. For example, fitting a very high-degree polynomial to noisy data yields low bias (fits training well) but huge variance (fails on new data), whereas fitting a straight line to data that's actually quadratic yields high bias (misses curvature) but low variance. Good model design seeks a balance: enough complexity to capture real structure (low bias) without over-sensitivity to noise (low variance).

Overfitting and Underfitting

These concepts are closely tied to bias and variance. **Underfitting** occurs when a model is too simple to capture the underlying patterns. It yields poor accuracy on training data (high training error) and thus also on any test data. For instance, using a linear model on clearly non-linear data will underfit, as the model cannot learn the true relationships. In practice, underfit models show large errors on both training and validation sets.

In contrast, **overfitting** happens when a model is too complex relative to the amount of training data. The model "memorizes" the training examples, capturing even random noise. An overfit model will have very low training error but much higher error on new (validation/test) data. For example, a decision tree grown without depth limit will often fit the training labels perfectly but then fail to generalize. In overfitting, the decision boundary becomes overly wiggly or irregular, fitting idiosyncrasies of the training set rather than true signal. In summary: overfitting models score low error on training but high error on unseen data, while underfitting models perform poorly on both training and testing data.

Train/Validation/Test Splitting

To reliably assess a model's performance and control overfitting, we split our data into (at least) three sets:

- **Training Set:** The subset used to train the model. The model's parameters (e.g. weights in regression, or tree splits) are fit on this data. In each training epoch, the model updates to better fit this data. A larger training set generally yields better learning (reducing variance).

- **Validation Set:** A held-out set used during training to tune hyperparameters (e.g. regularization strength, tree depth, K in KNN) and monitor for overfitting. After each epoch or in cross-validation folds, we evaluate the model on the validation set. If validation error starts rising while training error falls, it's a sign of overfitting. We adjust hyperparameters accordingly. Splitting out a validation set helps ensure the model and its tuning are not biased toward the training data.

- **Test Set:** A final held-out set, used only after training and tuning are complete, to evaluate the model's generalization. The test error is an unbiased measure of how the model will perform on new data. It is crucial that the model has never "seen" this data during training or validation to ensure a fair assessment.

A common practice is to split data roughly 60–80% training, 10–20% validation, and 10–20% test, though exact ratios depend on dataset size. By separating data this way, we avoid the pitfall of testing on the same data we trained on (which would give an overly optimistic accuracy). In practice, tools like k-fold cross-validation can also be used to make efficient use of data, but the principle remains: **never use test data for training or tuning.**

Data Preprocessing

Before feeding data to a model, we usually preprocess it:

- **Handling Missing Values:** Real-world data often has gaps (NaNs, blanks, "NA" values). Missing values can bias results or reduce effective sample size if ignored. Common strategies include deleting rows/columns with missing entries (if few) or imputing values (e.g. replacing with the mean or median of that feature, or using model-based imputation). For example, if a "height" feature has missing entries, one might fill them with the average height. Proper handling of missing data is important to avoid introducing bias or reducing model accuracy.

- **Feature Scaling:** Many algorithms (e.g. SVM, KNN, gradient descent-based models) perform better when numeric features are on similar scales. If one feature ranges 0–1000 and another 0–1, the large-scale feature can dominate the learning. Standardization (subtracting mean and dividing by standard deviation) or normalization (scaling to a fixed range, e.g. 0–1) is often applied. For instance, scikit-learn's StandardScaler transforms features to zero mean and unit variance. After scaling, a feature with originally large magnitude no longer skews the optimization. This step is especially crucial for algorithms that use distance or dot products (e.g. SVM, KNN, logistic regression).

- **Encoding Categorical Features:** Machine learning algorithms generally require numeric inputs. Categorical features (like country names or color labels) must be converted. Two common methods are **label encoding** (assigning each category an integer) and **one-hot encoding** (creating binary dummy variables). Label encoding maps categories to arbitrary integers (e.g. “red”→0, “green”→1, “blue”→2), which is simple but can misleadingly impose an order. One-hot encoding creates a separate binary column for each category (e.g. a “Color_red” column that is 1 if color is red, else 0), which avoids implying order. For example, one-hot encoding of a “Country” column with values {“US”, “FR”, “JP”} creates three new binary features. The choice depends on the model and data: tree-based models can handle integer labels reasonably, but linear models often prefer one-hot encoding of nominal categories. In all cases, categorical encoding converts qualitative data into a numeric form the model can use.

SageMaker Built-In Algorithms

Amazon SageMaker provides a range of built-in algorithms that implement common machine learning methods. These are optimized for scalability and can be launched easily from SageMaker Studio. Key examples include:

- **Linear Learner:** Implements both linear regression and linear classification (logistic regression) in one framework. It learns weights for features to minimize loss, optimizing either a continuous loss (like least squares) or a classification loss (like logistic). SageMaker’s documentation notes that Linear Learner “learns a linear function for regression or a linear threshold function for classification”. In practice, you specify `predictor_type='regressor'` or `'binary_classifier'` (or `'multiclass_classifier'`) when using it. Linear Learner also supports validation data to choose the best model. It is convenient for tabular data and can automatically explore different objective functions and regularizations.

- **XGBoost:** A popular gradient-boosted trees algorithm, optimized for speed and performance. SageMaker includes a built-in XGBoost implementation. XGBoost builds an ensemble of decision trees sequentially, each correcting errors of the previous ones. It works well for many structured (tabular) datasets. The SageMaker docs describe XGBoost as “an efficient open-source implementation of the gradient boosted trees algorithm” that can be used for regression, binary classification, and multiclass classification. By tuning hyperparameters (tree depth, learning rate, regularization, etc.) one can balance bias and variance. SageMaker’s XGBoost is pre-packaged so you simply point it at your data (e.g. in S3) and it handles distributed training if needed.

- **BlazingText:** A specialized algorithm for text data. It implements Word2Vec (to learn word embeddings) and a supervised text classifier (based on Facebook’s FastText) in highly optimized C++/CUDA. BlazingText can train on huge text corpora quickly, producing word embeddings or directly performing text classification. The SageMaker docs explain that BlazingText provides “highly optimized implementations of the Word2vec and text classification algorithms” for scalable NLP tasks. For example, in its **supervised mode** it can learn to classify sentences by training a neural model (utilizing GPUs), and in **unsupervised mode** it can learn continuous word vectors (skip-gram or

CBOW). Data for BlazingText must be tokenized text (one sentence per line) with optional label prefixes for supervised use.

- **Image Classification:** SageMaker offers built-in deep learning models for image tasks. For example, the MXNet Image Classification algorithm is a CNN-based model for multi-label image classification. It takes raw images (JPEG/PNG or RecordIO format) and learns to assign one or more class labels per image, using convolutional neural networks. It supports training from scratch or fine-tuning pre-trained networks when data is limited. In SageMaker, one points the Image Classification estimator to an S3 bucket of labeled images (often organized in folders or via manifest files) and specifies model hyperparameters (like number of layers, epochs, etc.). SageMaker also offers a TensorFlow-based Image Classification (using pretrained models) and other vision algorithms (object detection, etc.) with transfer learning support.

- **K-Nearest Neighbors (built-in):** SageMaker even has a KNN algorithm. As described in SageMaker's built-in list, "K-Nearest Neighbors (k-NN) Algorithm – a non-parametric method that uses the k nearest labeled points to assign a value.". This implementation accelerates neighbor searches (often via specialized data structures) to scale to larger datasets.

Each SageMaker built-in algorithm is optimized for distributed training, automatic data handling, and logging. For example, these algorithms support both file-mode and pipe-mode input (streaming from S3), and automatically output model artifacts and metrics to S3/CloudWatch. When using SageMaker Studio, you typically select one of these built-ins (via the SDK or the built-in algorithm recipes) to train your model quickly without writing low-level training code.