MODULE 3 · STUDY NOTES

Implementing a Machine Learning Pipeline with Amazon SageMaker

The end-to-end, iterative ML pipeline — framing a business problem, then collecting, preparing, training, hosting, evaluating, and tuning a model with Amazon SageMaker

This module walks a machine learning problem end to end through a repeatable **ML pipeline**, using Amazon SageMaker as the workbench. The pipeline is a loop, not a line: you **formulate the problem**, **collect and secure** the data, **evaluate** and **preprocess** it, **engineer features**, **select and train** a model, **deploy** it, **evaluate** its accuracy, and **tune** it — then circle back with more or better features and data until the results meet the business goal. The module focuses on **supervised learning** (a running credit-card-fraud example plus wine, car, and vertebral-column datasets), but the process adapts to other ML types. Along the way you meet the AWS services that power each stage — Amazon S3 and AWS Glue for data, the open-source pandas library for exploration, SageMaker built-in algorithms such as XGBoost for training, SageMaker hosting and batch transform for deployment, the confusion-matrix metrics for evaluation, and SageMaker automatic model tuning and Autopilot for optimization.

LEARNING OBJECTIVES

- Formulate a problem from a business request
- Obtain and secure data for machine learning (ML)
- Build a Jupyter notebook using Amazon SageMaker
- Outline the process for evaluating data
- Explain why data needs to be preprocessed
- Use open source tools to examine and preprocess data
- Use Amazon SageMaker to train and host an ML model
- Use cross-validation to test the performance of an ML model
- Use a hosted model for inference
- Create an Amazon SageMaker hyperparameter tuning job to optimize a model's effectiveness

3.1 The ML pipeline and problem formulation

The **machine learning pipeline** is the spine of this module. Its stages — problem formulation, collect and label data, evaluate data, feature engineering, select and train the model, deploy the model, evaluate the model, and tune the model — map one-to-one onto the module's eight sections. The pipeline is **iterative**: after evaluating, if the model does not *meet the business goal*, you loop back with **feature augmentation** or **data augmentation** and try again. On a real-world problem you might iterate many times before you reach a solution the business accepts.

Everything starts with the **business objective**. You cannot measure a solution's performance without a clear goal, and you frequently must clarify the business problem before you target a solution. Interview the business and the domain experts before writing any code.

QUESTIONS THAT DEFINE THE BUSINESS OBJECTIVE

- How is this task done today, and how will the business **measure success**?
- How will the solution be used, and what **assumptions** have been made?
- Do **similar solutions** already exist that you might learn from?
- **Who are the domain experts** who understand the data and the problem?

With the goal clear, **frame it as an ML problem**. First ask whether machine learning is even the right tool or whether a traditional approach makes more sense. If it is an ML problem, decide whether it is **supervised or unsupervised** (do you have labeled data?), identify the **target to predict**, confirm you have **access to the data**, set a **minimum acceptable performance**, consider how you would solve it **manually**, and reach for the **simplest** solution.

WORKED EXAMPLE – FRAMING CREDIT-CARD FRAUD

The problem: *identify fraudulent credit-card transactions so you can stop them before they process.* The business **why**: reduce the number of customers who cancel their membership because of fraud. Making it **measurable**: a successful outcome is a *10 percent reduction in customers who file fraud claims within a 6-month period* — a quantitative statement you can track. Turning it into a **model**: the model outputs whether a transaction is *fraudulent or not fraudulent*. Because you hold historical data with fraud labels, this is **supervised learning**, and because the output is one of two classes, it is a **binary classification** problem.

TABLE 3.1 The module's three datasets (UC Irvine ML Repository)

DATASET	QUESTION IT ASKS	WHAT YOU PRACTICE
Wine quality	Can the composition of the wine predict its quality and therefore its price?	View statistics, handle outliers, scale numeric data
Car evaluation	Can a car's attributes predict whether it will be purchased?	View statistics, encode mostly text (categorical) data
Vertebral column	Do the biomechanical features indicate an abnormality (disk hernia or spondylolisthesis)?	The full end-to-end lab: evaluate, encode, train, and tune

COMMON PITFALL

Do not skip framing. Ask **why** enough times to reach a solid problem statement, and insist that the outcome can be **measured** — a qualitative wish ("catch fraud") is not a target until it becomes quantitative ("10% fewer fraud claims in 6 months"). Most problems then resolve to **classification** (binary or multiclass) or **regression** (a numeric prediction).

3.2 Collecting and securing data

The first real step of the pipeline is to **obtain the data** that will train your model to reach the intended output. Three questions frame it: How much data do you have, and where is it? Do you have **access** to it? And what solution can bring it all into one **centralized repository**? Data is typically scattered across many systems, so consolidating it is a project in itself.

TABLE 3.2 Where training data comes from

SOURCE	WHAT IT IS	EXAMPLES
Private data	Data you or your customers already hold in existing systems	Log files, customer invoice databases
Commercial data	Data a commercial entity collected and sells access to (AWS Data Exchange, AWS Marketplace)	Reuters, Change Healthcare, Dun & Bradstreet, Foursquare
Open-source data	Publicly available datasets — check for usage limits	Kaggle, WHO, U.S. Census Bureau, NOAA, UC Irvine, AWS

ANATOMY OF ML DATA

- Supervised ML needs many **observations** — rows for which the target answer is already known. This is **labeled data**.
- Each observation has two parts: the **target** (the answer to predict, such as fraud or not fraud) and one or more **features** (attributes such as date, vendor, or amount).
- Labels often arrive *after the fact* — for fraud, the true label is recorded once the cardholder reports the charge, then used to train future models.
- Bring in a **domain expert** to help decide which features and target the model actually needs.

Data must be **representative** of what the model will see in production. To predict fraud you need both **positive** (fraudulent) and **negative** (legitimate) examples so the algorithm can learn the patterns that separate them. If real fraud runs at about 3 percent but your training set contains only 0.4 percent fraud, the model may fail to learn the fraudulent pattern — a **class imbalance** problem.

TABLE 3.3 AWS services for finding and storing data

SERVICE	WHAT IT OFFERS
Amazon S3	Object storage; store any amount of data, individual objects up to 5 TB, redundant across facilities; event notifications can trigger Lambda
Amazon FSx for Lustre	Serves S3 data to SageMaker at high speed; copies once, then reuse across training iterations to avoid repeated downloads
Amazon EFS	File system you can train directly from — no data movement, faster start times; handy for data scientists iterating from notebooks
Amazon RDS	Managed relational database service
Amazon Redshift	Managed data warehouse service
Amazon Timestream	Managed time-series database built for large volumes of IoT data

To pull scattered data together you run an **ETL** process. **Extract** copies data from the sources to one location; **Transform** modifies, combines, or reshapes records as needed; **Load** writes the result into a repository such as Amazon S3 or Amazon Athena. A typical ETL framework has a **crawler** (connects to a data store, infers the schema, and writes metadata tables to a catalog), a **job** (the business logic that does the work), and a **schedule or event** that runs it periodically.

AWS Glue is a fully managed, **serverless** ETL service — no infrastructure to set up — that makes it simple to categorize, clean, enrich, and move data between stores. It centers on the **AWS Glue Data Catalog** (a central metadata repository), an ETL engine that automatically generates **Python or Scala** code, and a flexible scheduler that handles dependencies, monitoring, and retries. Glue works with semistructured data through a **dynamic frame** (like an Apache Spark DataFrame but self-describing, so no schema is required up front) and can run **event-driven** pipelines — for example, a Lambda function triggers a Glue job when new data lands in Amazon S3. Data scientists can also write their own extract-and-load scripts in a Jupyter notebook using the `boto3` AWS library.

The data must also be **secured** — a business and regulatory requirement, not an afterthought. Three levers protect ML data: **controlling access**, **encrypting** the data, and **auditing** activity. AWS CloudTrail supplies the audit trail, giving governance, compliance, and risk auditing across the account: it logs, monitors, and retains account activity, provides an **event history** of actions taken through the console, SDKs, CLI, and other services, and helps detect unusual activity.

TABLE 3.4 Securing your ML data – three levers

LEVER	WHAT IT DOES
Access control (IAM)	AWS Identity and Access Management controls who can reach a resource — for example, a policy that allows only `s3:GetObject` (read) on one bucket for one role
Encryption	Turn on Amazon S3 default encryption and Amazon RDS encryption; protect data at rest and, via SSL/TLS, in transit
Audit (AWS CloudTrail)	Logs user activity and API usage, provides an event history for security analysis and change tracking, and helps detect unusual activity

> **COMMON PITFALL**
>
> Course datasets are public, but real customer or health records must be secured. Securing data is **both** controlling access (IAM) **and** encrypting it — not one or the other — with CloudTrail on top for the audit trail that regulated industries require.

3.3 Evaluating your data

Before you can run statistics, the data must be in the **right format** for analysis. SageMaker algorithms support training with **CSV**, and most exploration tools read CSV too. Data must generally be **numeric** for algorithms to use it (text conversion comes later, in feature engineering). You also need at least some **domain knowledge**: to model whether symptoms indicate a disease, you must understand how the symptoms relate to the disease.

The open-source **pandas** library is the workhorse for this stage. It reformats CSV, JSON, Excel, Pickle, and other formats into a tabular **DataFrame** — think of a spreadsheet or SQL table with instances (rows) and attributes (columns). Each column is a **series** (a one-dimensional labeled array). Useful moves: `read_csv()` to load; the `shape` attribute for row/column counts; `columns` and `index` for labels; `info()` or `dtypes` to check column types; and `astype()` to fix a type (a numeric column can arrive as text because of missing data or a stray value).

THREE KINDS OF DESCRIPTIVE STATISTICS

- **Overall statistics** — the number of rows (instances) and columns (features); too many features means high **dimensionality**, which can hurt performance.
- **Attribute statistics** — for numeric columns: mean, standard deviation, variance, minimum, and maximum, which describe an attribute's shape.
- **Multivariate statistics** — relationships *between* variables, such as correlations; highly correlated features together can keep model loss from converging.

Mean vs. median matters when data is skewed. The mean describes a symmetrical distribution well, but large **outliers** pull the mean away from the true center; the **median** resists them and better captures central tendency. pandas exposes `describe()` for a full summary, plus `mean()`, `median()`, `min()`, `max()`, and `skew()` on a column, `value_counts()` to count categories, and `groupby()` to compute a statistic per category.

For **categorical** attributes, look at the **frequency** of each value. In the car dataset, `describe()` shows the `class` column's top value `unacc` accounting for 1,210 of 1,728 rows — about 70 percent — which signals a possible **class imbalance**. When the imbalanced column is the target (as with a fraud rate of a tenth of a percent), the algorithm may not learn to predict the rare class.

TABLE 3.5 Visualizations for understanding data

VISUALIZATION	WHAT IT REVEALS
Histogram	Distribution shape of one feature — is it normal, how many peaks, is it skewed (values are binned; taller bars are common values)
Density and box plots	Range, peak, and outliers of a numeric feature; a box plot spans Q1–Q3 with the median at Q2 and whiskers at 1.5×IQR, points beyond are outliers
Scatter plot / scatter matrix	Relationship between two (or pairwise among several) numeric variables; spots trends and multivariate outliers
Correlation matrix / heat map	Linear relationships from +1 to −1; a Seaborn heat map colors strong positive (dark green) and negative (dark brown) correlations

COMMON PITFALL

A correlation of **zero** means no *linear* relationship — not that no relationship exists at all. And **high** correlation between two features is not automatically good: closely correlated features used together can degrade the model (loss may not converge). Watch for both.

3.4 Feature engineering

Feature engineering makes models more successful through two activities. **Feature selection** keeps the features most relevant to the target and discards the rest — removing redundancy and irrelevance and limiting feature count to prevent **overfitting**. **Feature extraction** builds valuable new information from raw data by reformatting, combining, and transforming primary features into new ones the model can consume.

Feature extraction spans many tasks: fixing invalid values, wrong formats, misspellings, duplicates, and inconsistency; rescaling and **normalization**; **encoding categories**; removing or reassigning **outliers**; and bucketing, decomposition, aggregation, combination, transformation, and dimensionality reduction. Many of these are ordinary data-wrangling chores; the ML-specific ones are converting text to numbers, handling outliers, and rescaling.

ENCODING ORDINAL DATA	ENCODING NON-ORDINAL DATA
order is meaningful — preserve it ▪ Categorical data is non-numeric and must be encoded to a numeric scale ▪ When categories have a natural order, map them to ordered numbers ▪ Maintenance cost: Low→1, Medium→2, High→3, Very High→4 ▪ First make text uniform (for example, Med and Medium)	*no order — do not invent one* ▪ If there is no order, do **not** assign 1, 2, 3… ▪ That would imply, for example, blue outranks red ▪ Split into **multiple columns**, one per category (like check boxes) ▪ Colors red/blue/green become three columns holding 1 or 0

TABLE 3.6 Common kinds of dirty data to clean

PROBLEM	EXAMPLE	ACTION
Variations in strings	Med. vs. Medium	Convert to standard text
Variations in scale	Number of doors vs. number sold	Normalize to a common scale
Multiple items in one column	"Safe high-maintenance"	Parse into multiple columns
Missing data	Missing values in a column	Delete rows, or impute values
Outliers	Abnormally distant values	Delete, transform, or impute

Missing data makes it hard to interpret the relationship between a feature and the target; causes include undefined values and data collection or cleaning errors. Most ML algorithms cannot handle missing values automatically, so you must act. Find them with pandas: `df.isnull().sum()` counts per column and `df.isnull().sum(axis=1)` counts per row. Before you drop or impute, ask *why* the data is missing, whether it is missing at random, and how much of the dataset it represents — **impute** when values are few and random, but **drop** the row or column when a large share is missing.

You can **drop** missing values with `df.dropna()` (rows), `df.dropna(axis=1)` (columns), or a subset. To **impute**, replace missing values with the mean, median, or most frequent value (mean or median for numeric); Scikit-learn's imputer does this **univariate** (one row) or **multivariate** (many rows), and advanced options include K-nearest-neighbor and multiple imputation. **Outliers** — points at an abnormal distance from the rest — can add richness but hurt accuracy; find them with box plots (univariate) or scatter plots (multivariate) and handle them by **deleting** (if caused by an artificial error), **transforming** (for example, taking the natural log to shrink the extreme), or **imputing** a replacement such as the feature mean.

TABLE 3.7 Feature selection methods

METHOD	HOW IT WORKS	EXAMPLES / TRADE-OFF
Filter	Uses statistical measures of a feature's relevance to the target — no model training	Pearson's correlation, LDA, ANOVA, chi-square; fast and general, but not tuned to a model
Wrapper	Trains a model on each feature subset and scores it on a hold-out set	Forward and backward selection; best-performing set, but compute-intensive with overfitting risk
Embedded	Algorithm has built-in feature selection; blends filter and wrapper qualities	Decision trees, LASSO and RIDGE (penalties reduce overfitting)

COMMON PITFALL

The single most common feature-engineering error: encoding **non-ordinal** categories as 1, 2, 3…. The model reads the numbers as magnitudes and infers an order that isn't there. Non-ordinal data belongs in **one column per category** (one-hot style), never on a single numeric scale.

3.5 Training

Clean data may still not be ready to train — algorithms expect specific **file formats**. For SageMaker, **CSV** files must have **no header record** and the **target variable in the first column**, with features to its right. Most SageMaker algorithms work best with the optimized **protobuf recordIO** format, which converts each instance into 4-byte floats. RecordIO enables **Pipe mode**, which *streams* data directly from Amazon S3 for faster start times, better throughput, and smaller EBS volumes; the alternative **File mode** copies the whole dataset onto the training instance's volume first.

You must **split** the data because evaluating a model on the same data it trained on causes **overfitting** — the model memorizes particulars instead of learning relationships, and then fails on new data. The **holdout method** splits data into **training** (features and labels feed the algorithm), **validation** (where you notice what to tweak and tune), and **test** (features only — the model predicts the labels, and this performance estimates production). Common splits are **80/10/10** or, with more data, **70/15/15** (train/validation/test).

CROSS-VALIDATION AND SHUFFLING

- **K-fold cross validation** suits small datasets: partition the data into K segments, train on the rest and validate on each segment in turn, then **average** the results — so all data is used and models can be compared.
- A 5-fold example trains and tests five times on different chunks until every data point has been tested.
- **Shuffle** first: data left in order (for example, sorted by the target) biases the model and can hide target classes from training.
- **Stratified sampling** keeps each split's class ratios close to the full set; `train_test_split` from scikit-learn with `stratify` does this.

Amazon SageMaker offers four ways to train. Built-in algorithms run from the console, AWS CLI, a notebook, or the SageMaker Python SDK (containers are used behind the scenes). **Supported frameworks** ship as prebuilt containers for Apache MXNet, TensorFlow, PyTorch, and Chainer, plus scikit-learn and SparkML — you supply your code as an entry point via an Estimator. **Custom frameworks** let you package your own script or algorithm in any language (for example, a team that builds models in R). **AWS Marketplace algorithms** are ready-to-use models from third parties.

TABLE 3.8 Amazon SageMaker built-in algorithms

CATEGORY	ALGORITHMS	SOLVES
Classification	XGBoost, Linear learner	Binary or multiclass labels
Quantitative (regression)	XGBoost, Linear learner	Predict a numeric value
Recommendations	Factorization machines	Recommendations, time-series
Unsupervised	K-means, PCA	Groupings; dimensionality reduction
Specialized	Image classification, Seq2Seq, NTM, LDA	Images and deep learning tasks

THREE ALGORITHMS WORTH KNOWING BY NAME

- **XGBoost** — open-source **gradient boosted trees**; combines an ensemble of weaker models, handles varied data and many hyperparameters, and fits regression, classification (binary and multiclass), and ranking.
- **Linear learner** — solves both classification and regression, and explores several training objectives at once to pick the best from your validation set.
- **K-means** — unsupervised; finds discrete groupings where members are as similar as possible, using the averaging of data to locate each group's center.

To train, you create a **training job** that specifies four things: the **S3 URL of the training data**; the **compute resources** (ML instances mixing CPU, GPU, memory, and networking, sized to the workload); the **S3 URL for the job's output**; and the **Amazon ECR path** where the training code (container image) is stored.

COMMON PITFALL

The most-tested SageMaker training detail: a **CSV** training file for SageMaker has **no header** and the **target in the first column**. Remember it as "label first, no header" — the opposite of a typical spreadsheet export.

3.6 Hosting and using the model

After a model is **trained, tuned, and tested**, it is ready to deploy. The stages can look out of order because testing a model requires making inferences, which typically requires **deployment** — deployment for testing uses the same mechanics as production, just at smaller scale. SageMaker hosts models for anything from a few requests to tens of thousands.

SAGEMAKER HOSTING SERVICES	BATCH TRANSFORM
for single, real-time predictions	*for an entire dataset at once*
▪ Deploys compute instances behind a **load-balanced** HTTPS endpoint ▪ Applications call the endpoint API to get predictions ▪ Scale instances up or down based on demand ▪ **Single-model** endpoint for one model; **multi-model** shares a container across many	▪ No permanent endpoint to deploy and maintain ▪ SageMaker spins up the model and predicts the whole dataset ▪ Stores results in Amazon S3, then terminates the compute instances ▪ Great at test time — run the full validation set with no glue code

The **goal of deployment** is a managed environment that hosts the model to serve inference **securely and with low latency**. Deploying with SageMaker means auto-scaling ML instances **across multiple Availability Zones**: you specify the instance type and the minimum and maximum counts, and SageMaker launches the instances, deploys the model, and sets up a secure **HTTPS endpoint**. Because model changes no longer require application code changes, you can integrate new models in minutes. SageMaker also manages the production infrastructure — health checks, security patches, and routine maintenance — with built-in **Amazon CloudWatch** monitoring and logging.

Multi-model endpoints cut cost by hosting many models in a shared serving container, loading models into memory and scaling them to traffic — better endpoint utilization than one endpoint per model. **Inference pipelines** solve a consistency problem: the *same data-processing steps used in training must be applied to every inference request*, or predictions come out wrong. A pipeline reuses the training preprocessing code (no second copy to maintain) by running a sequence of containers collocated on the same EC2 instances, so feature processing and inference run together with low latency.

> **COMMON PITFALL**
>
> Deploying is a **continuous process, not a one-time exercise**. Once the model is in production, monitor the production data and **retrain** when needed — new data accumulates over time and can reveal alternative or new outcomes the current model has never seen.

3.7 Evaluating the accuracy of the model

Evaluation determines how well the model will predict the target on **future** data. Since future targets are unknown, you assess performance on **held-out** data whose answers you know and treat that as a proxy for production. Crucially, the **model metric must align with the business metric** you defined during problem formulation — the two are usually highly correlated — and the type of ML problem (classification vs. regression) shapes which metric you choose.

For classification, start with the **confusion matrix**, which compares predicted vs. actual classes. Using a cat / not-cat model: a **true positive (TP)** predicts cat and it is a cat; a **true negative (TN)** predicts not-cat and it is not a cat; a **false positive (FP)** predicts cat but it is not (a positive but incorrect prediction); a **false negative (FN)** predicts not-cat but it is a cat. TP and TN are the good outcomes.

TABLE 3.9 Metrics derived from the confusion matrix

METRIC	DEFINITION	REACH FOR IT WHEN
Sensitivity (recall, TPR)	TP ÷ all actual positives — share of positives caught	A false negative is costly (terminal illness)
Specificity (TNR)	TN ÷ all actual negatives — share of negatives caught	Correctly clearing negatives matters
Precision	TP ÷ (TP + FP) — positive predictions that are correct	A false positive is costly (spam filter)
Accuracy (score)	(TP + TN) ÷ all — overall correctness	Classes balanced; weak when TNs dominate
F1 score	Balances precision and sensitivity in one number	Class imbalance, keeping the two equal

WHICH MODEL IS BETTER? IT DEPENDS ON THE GOAL

Two models score the same test data. **Model A** — TP 107, FP 23, FN 69, TN 42 — gives **sensitivity 60%** (107 of 176 cats) and **specificity 64%** (42 of 65 not-cats). **Model B** — TP 148, FP 53, FN 28, TN 12 — gives **sensitivity 84%** but **specificity 18%**. If the goal is to catch as many cats as possible and false positives are acceptable, choose **B**. If correctly identifying not-cats matters, **A** is safer. There is no single "better" model — only the model that fits the business trade-off, which is why diagnosing heart disease and rating a fun website call for different choices.

Classification models actually return a **probability** between 0 and 1; a **threshold** (often 50 percent, but adjustable) converts it to a class, and moving the threshold trades sensitivity against specificity. A **receiver operating characteristic (ROC)** graph plots sensitivity against the false-positive rate (1 − specificity) at every threshold — the closer the curve hugs the top-left corner, the better, while the diagonal represents random guessing. To compare whole models at a glance, use **AUC-ROC**, the area under that curve: higher is better.

Accuracy — correct predictions divided by all predictions — is widely used but weak when **true negatives dominate**: a fraud model can post high accuracy while barely catching fraud. That is why **precision** (which ignores negatives) suits high-false-positive-cost cases like spam, **sensitivity** suits high-false-negative-cost cases like disease, and the **F1 score** combines both when you have class imbalance but want to keep precision and recall balanced. For **regression**, use **mean squared error (MSE)** — square the difference between prediction and actual for every observation and sum them — or **R-squared** for linear models; scikit-learn's metrics library provides them directly.

Evaluation feeds the **tuning loop**. With metrics in hand you might select a different set of features and retrain, use different data with the same features (recall k-fold cross validation), or tune the model's own **parameters** — the subject of the next section. If the model does not meet the business goal, iterate.

COMMON PITFALL

Never let a single high **accuracy** number settle a classification model, especially with imbalanced classes. A model that is 99% accurate on data that is 99% negatives may catch almost no positives. Check sensitivity, precision, F1, and AUC-ROC against the *business* cost of each error type.

3.8 Hyperparameter and model tuning

Hyperparameters are the knobs that tune the algorithm to improve performance — set before training rather than learned from data. They fall into three categories, and knowing the category helps you decide what to adjust.

TABLE 3.10 Categories of hyperparameters

CATEGORY	GOVERNS	EXAMPLES
Model	The architecture that defines the model itself	Number of layers, activation functions, filter size, pooling, stride, padding
Optimizer	How the model learns patterns from data	Gradient descent, stochastic gradient descent, Adam (momentum), Xavier or He initialization
Data	Attributes of the data itself; useful for small or homogeneous datasets	Data augmentation such as cropping or resizing images

Traditionally, tuning was **manual**: someone with domain experience picked hyperparameters by intuition, trained, scored on validation data, and repeated until satisfied. It works, but it is rarely thorough or efficient — for complex systems like deep neural networks, exploring every combination is impractical.

Amazon SageMaker automatic model tuning (hyperparameter tuning) finds the best version of a model by running many **training jobs** over the algorithm and **ranges of hyperparameters** you specify, then choosing the values that perform best on the **objective metric** you pick. It uses **Gaussian Process regression** to predict which values might improve fit and **Bayesian optimization** to balance exploring new values against exploiting good ones. It works with built-in algorithms, prebuilt deep learning frameworks, and bring-your-own-algorithm containers.

WORKED EXAMPLE – TUNING A FRAUD CLASSIFIER

You want to solve a **binary classification** problem on a fraud dataset and **maximize AUC** by training a **linear learner** model, but you do not know the best values for `learning_rate`, `beta_1`, `beta_2`, and `epochs`. You specify a **range** for each, and SageMaker runs training jobs across those ranges to find the combination that scores highest on your objective metric — then returns the training job with the **highest AUC**.

TUNING BEST PRACTICES

- Do **not** adjust every hyperparameter — tune only the few most likely to matter.
- **Limit the ranges** to values that are most effective; if the best results cluster in part of a range, narrow to that part.
- Run **one training job at a time** rather than many in parallel — tuning improves through successive rounds, so serial usually wins with less compute.
- In **distributed** training jobs, make sure the objective metric you want is the one reported back (tuning takes the last-reported metric).
- Convert **log-scaled** hyperparameters to **linear-scaled** yourself when you can, to help the optimizer.

Amazon SageMaker Autopilot accelerates the whole cycle. You create a job supplying the training data, test data, and target; Autopilot analyzes the data, selects appropriate features, and trains and tunes models — documenting the metrics and returning the **winning model, its metrics, and a Jupyter notebook** you can use to investigate the results. It does not remove the need to preprocess your data, but it saves substantial time on feature selection and model tuning.

COMMON PITFALL

Tuning **does not always improve** the model — the deck says so explicitly. Treat hyperparameter tuning as part of the scientific method, not a guaranteed win, and resist the urge to run many parallel jobs over wide ranges; a focused, serial search over a few well-chosen hyperparameters usually beats it.

Module summary

- The ML pipeline is iterative: problem formulation → collect & label data → evaluate data → feature engineering → select & train → deploy → evaluate → tune, looping back with feature/data augmentation until the model meets the business goal.
- Problem formulation converts a business request into an ML problem — ask why until the goal is solid, make success measurable, and classify it as classification (binary/multiclass) or regression.
- Supervised learning needs many labeled observations (target + features) that are representative of production; collect from private, commercial, and open-source sources, consolidate via ETL/AWS Glue, and store in services like Amazon S3, RDS, Redshift, or Timestream.

- Secure data three ways: control access with IAM, encrypt at rest and in transit (S3/RDS encryption, SSL/TLS), and audit with AWS CloudTrail.
- Evaluate data by first getting it into numeric CSV form, then exploring with pandas — DataFrames, descriptive statistics (overall, attribute, multivariate), and visualizations (histogram, box plot, scatter, correlation heat map).
- Feature engineering = feature selection (keep relevant features; filter, wrapper, embedded methods) plus feature extraction (encode categoricals — ordinal vs. non-ordinal — and clean missing data and outliers).
- Prepare training data as CSV (no header, target first column) or protobuf recordIO (enables Pipe-mode streaming from S3); split with holdout (80/10/10 or 70/15/15) or k-fold cross validation, shuffling/stratifying to avoid order bias and overfitting.
- SageMaker trains via built-in algorithms (XGBoost, Linear learner, K-means, PCA, Factorization machines, and specialized algorithms), supported/custom frameworks, or AWS Marketplace — launched through a training job (S3 data, compute, S3 output, ECR code path).
- Deploy with SageMaker hosting services (real-time, load-balanced HTTPS endpoint; single- or multi-model) or batch transform (whole dataset, then terminate); use inference pipelines for consistent preprocessing, and keep monitoring and retraining.
- Evaluate classification with the confusion matrix and its metrics — sensitivity, specificity, precision, accuracy, F1, and AUC-ROC — choosing the metric that matches the business cost; evaluate regression with MSE and R-squared.
- Tune hyperparameters (model, optimizer, data) with SageMaker automatic model tuning (Gaussian process + Bayesian optimization) over specified ranges, following best practices, and use Autopilot to accelerate feature selection and tuning.

Review questions

1. What two things must be true before a business request becomes a workable ML problem, and what two problem types usually result?

Answer: You must have asked why enough to reach a solid problem statement, and you must be able to measure the outcome (move from qualitative to quantitative). Most problems then resolve to classification (binary or multiclass) or regression (a numeric prediction).

2. What are the two elements of an observation, and why must training data be representative?

Answer: The target (the answer to predict) and the features (attributes used to find patterns). Data must be representative because the model needs both positive and negative examples in realistic proportions — too few of the rare class (for example, 0.4% fraud when reality is 3%) and it can't learn to predict it.

3. What is ETL, which AWS service runs it serverlessly, and what are the three ways to secure data?

Answer: ETL is extract, transform, load — pulling data from many sources, reshaping it, and loading it into a repository such as Amazon S3. AWS Glue runs ETL serverlessly with crawlers and a Data Catalog. Secure data by controlling access (IAM), encrypting it at rest and in transit, and auditing access with AWS CloudTrail.

4. Why must you split data before evaluating, and how do the holdout method and k-fold cross validation differ?

Answer: Evaluating on training data causes overfitting (the model memorizes instead of generalizing). Holdout splits into fixed training/validation/test sets (e.g., 80/10/10 or 70/15/15). K-fold cross validation partitions into K segments and rotates which is validation, averaging results — better for small datasets because all data is used.

5. How do you encode ordinal versus non-ordinal categorical data, and why does it matter?

Answer: Ordinal data (Low, Medium, High, Very High) maps to ordered numbers (1, 2, 3, 4) that preserve the order. Non-ordinal data (colors) must be split into multiple columns, one per category (1/0), because assigning single numbers would invent a false order the model would misread as magnitude.

6. From a confusion matrix, define precision and sensitivity, and explain when accuracy is a poor metric.

Answer: Precision = TP ÷ (TP + FP), the share of positive predictions that are correct (good when false positives are costly). Sensitivity/recall = TP ÷ all actual positives, the share of positives caught (good when false negatives are costly). Accuracy is poor when true negatives dominate — a high score can hide a weak ability to catch the rare positive, as in fraud detection.

7. What are the two SageMaker deployment options, and how do single- and multi-model endpoints differ?

Answer: Hosting services provide a persistent, load-balanced HTTPS endpoint for single real-time predictions; batch transform predicts an entire dataset, stores results in S3, and terminates the instances. A single-model endpoint hosts one model; a multi-model endpoint shares one serving container across many models for better utilization and lower cost.

8. Name the three categories of hyperparameters and describe SageMaker automatic model tuning.

Answer: Model (architecture: layers, activation, filter size, pooling, stride, padding), optimizer (how the model learns: gradient descent, SGD, Adam, initialization), and data (attributes such as augmentation). Automatic model tuning runs many training jobs over the ranges you specify, using Gaussian Process regression and Bayesian optimization to find the hyperparameter values that best score your chosen objective metric.