	Module/framework/package
1	Base R (in the stats library)
2	Big data version of R
3	Dask ML

4	**Spark R**
5	**Spark optimization**

6	Scikit-learn(Decision Tree)

Name and a brief description of the algorithm
The optimization method used in the glm function of R's stats package (version 3.6.2) is Iteratively Reweighted Least Squares (IWLS). As documented in the package, this algorithm is implemented in the default "glm.fit" function. IWLS transforms non-linear regression problems into a sequence of weighted least squares problems solved iteratively until convergence. The process begins with initial parameter estimates, then at each iteration it calculates a linear predictor from current parameters, computes fitted values using the inverse link function, determines observation weights based on the variance function of the specified family, calculates working residuals, and solves a weighted least squares problem to update parameters. The algorithm continues until convergence criteria are met or maximum iterations reached. IWLS effectively maximizes the likelihood for GLMs while handling the non-linearity introduced by various link functions. The implementation includes safeguards for boundary cases and allows for alternative optimization methods.
The h2o::glm() function uses the **L-BFGS algorithm**, a quasi-Newton optimization method, to estimate parameters of generalized linear models. L-BFGS approximates the Hessian matrix (second-order derivatives) using limited memory, making it highly efficient for large-scale problems with many parameters. This algorithm is well-suited for high-dimensional data and is further optimized in h2o through **parallelization and multithreading**, allowing it to process massive datasets quickly. It also supports **regularization techniques** like Lasso and Ridge (via elastic net), which improve model performance and prevent overfitting.
Alternating Direction Method of Multipliers (ADMM)- In Dask-ML, a key optimization method for fitting Generalized Linear Models (GLMs) like logistic and Poisson regression is the Alternating Direction Method of Multipliers (ADMM). ADMM is a powerful optimization algorithm particularly well-suited for large-scale and distributed data. It decomposes complex problems into smaller subproblems that are easier to solve and can be computed in parallel across multiple workers or machines. Each iteration alternates between minimizing an augmented Lagrangian function and updating dual variables, effectively coordinating local solutions to reach a global optimum. This makes ADMM ideal for scenarios involving big data or distributed computing environments, such as Dask arrays or dataframes. Dask-ML leverages this to efficiently handle out-of-core data, providing scalable GLM estimation

The spark.glm() function in SparkR uses the Iteratively Reweighted Least Squares (IRLS) algorithm to estimate the parameters of generalized linear models (GLMs). IRLS is an iterative optimization technique that maximizes the log-likelihood function by repeatedly solving a series of weighted least squares problems. In each iteration, it updates weights and residuals based on the current parameter estimates and the specified distribution family (e.g., Gaussian, Poisson, Binomial). This method continues until convergence criteria—such as a small change in deviance or coefficients—are met. In SparkR, IRLS is optimized for distributed computing, allowing it to efficiently process large datasets across a Spark cluster, providing scalability and improved performance for big data applications

Spark MLlib uses **Stochastic Gradient Descent (SGD)** and **Limited-memory BFGS (L-BFGS)** as core optimization algorithms to estimate the parameters of generalized linear models (GLMs).

SGD works by updating model parameters iteratively using gradients computed from randomly sampled subsets (mini-batches) of the training data. In each iteration, only a small portion of the data is used to compute a noisy but efficient estimate of the gradient, allowing faster updates and scalability for large datasets. Spark distributes this computation across worker nodes using RDDs, enabling parallel processing and improved performance on big data.

L-BFGS is a quasi-Newton method that approximates the Hessian matrix using gradients from previous iterations. Unlike SGD, it does not require explicit second derivatives but converges faster by incorporating curvature information. L-BFGS is especially effective for convex problems and is implemented in Spark as a low-level optimization primitive, typically paired with L2 regularization. It is recommended when higher accuracy and faster convergence are required.

Algorithm Name: Greedy Recursive Partitioning (CART - Classification and Regression Trees)

Description:
Decision Trees, including those in scikit-learn's DecisionTreeRegressor and DecisionTreeClassifier, use the CART (Classification and Regression Trees) algorithm, which applies a greedy, recursive partitioning method to optimize the tree structure. The goal is to minimize an impurity measure or loss function at each split in the tree. For regression problems (i.e., Generalized Linear Model analogs with continuous outputs), the impurity criterion is often Mean Squared Error (MSE), Mean Absolute Error (MAE), or Poisson deviance.

The algorithm proceeds by evaluating all possible splits for each feature and selecting the one that leads to the greatest reduction in impurity. This process is repeated recursively, creating branches until a stopping condition (e.g., max_depth, min_samples_split) is met. Although this is not an optimization in the mathematical sense (e.g., solving a convex problem), it greedily optimizes the split criterion locally at each node to approximate a globally optimal decision tree.

Because the full decision tree may overfit the training data, additional optimization is applied via pruning—such as cost-complexity pruning—to balance model complexity and generalization.

An example of a situation where using the provided GLM implementation provides superior performance compared to that of base R or its equivalent in Python (identify the equivalent in Python)
The direct equivalent to R's glm function in Python is the GLM class from the statsmodels package. This implementation provides similar functionality for fitting generalized linear models with various distributions and link functions. One situation where R's glm function from the stats package provides superior performance compared to its Python equivalent is when analyzing overdispersed count data with complex random effects. For example, when modeling ecological count data with many zero values and high variance, R's glm function with family=quasipoisson() handles overdispersion efficiently through its IWLS algorithm implementation. The iterative nature of IWLS in R's implementation is optimized for such cases, often converging faster and with better numerical stability.
Fitting a large-scale logistic regression model using the h2o package provides significant performance advantages over base R's glm() function. For example, in a scenario involving millions of observations and hundreds of predictor variables—such as customer click-through data in digital advertising—modeling can be highly computational. The h2o::glm() function is designed for in-memory distributed computing, allowing it to efficiently process large datasets that would typically exceed base R’s memory capacity. It uses a parallelized version of the L-BFGS optimization algorithm, enabling faster convergence by leveraging multithreaded execution. Compared to base R’s IRLS approach, L-BFGS in h2o delivers superior speed and scalability for big data applications.
A situation where Dask-ML’s GLM provides superior performance is when training a logistic regression model on large-scale clickstream data containing hundreds of millions of rows stored in a distributed file system. In this case, using base R’s glm() function or Python’s scikit-learn.LogisticRegression() would be inefficient or even infeasible, as both require the entire dataset to fit into memory on a single machine. Dask-ML, on the other hand, is designed to work with distributed data using Dask arrays and dataframes, allowing it to scale seamlessly across multiple cores or nodes. It leverages advanced optimization algorithms like ADMM and L-BFGS that are suitable for large-scale, out-of-core computations. As a result, Dask-ML’s GLM can handle much larger datasets efficiently, offering better performance, faster training times, and reduced memory usage compared to its R and scikit-learn equivalents in Python.

A situation where SparkR's spark.glm() provides superior performance is when fitting a Poisson regression model on a very large insurance dataset containing hundreds of millions of policy records stored in a distributed environment like HDFS. In this case, base R's glm() and Python's scikit-learn.PoissonRegressor() would struggle or fail because they require all data to fit into memory and run on a single machine. In contrast, spark.glm() is designed to work with distributed data using SparkDataFrames and runs across a Spark cluster. It uses a distributed version of the Iteratively Reweighted Least Squares (IRLS) algorithm, allowing it to process massive datasets efficiently and in parallel. As a result, SparkR enables scalable, fault-tolerant modeling that delivers better performance and speed compared to in-memory implementations in base R or scikit-learn when working with big data.

A situation where Spark's GLM implementation with optimization methods like Stochastic Gradient Descent (SGD) or L-BFGS provides superior performance is when training a logistic regression model on a massive e-commerce dataset containing hundreds of millions of user transactions across multiple product categories.

In this case, the dataset is stored in a distributed file system (like HDFS), and includes high-dimensional features such as user behavior, location, and product interactions. Attempting to train this model using base R's glm() or Python's scikit-learn.LogisticRegression() would fail or be extremely slow, as both rely on in-memory computation and cannot efficiently handle data that exceeds RAM.

Spark, however, distributes both data storage and computation across a cluster. Its GLM implementation, using distributed SGD or L-BFGS, scales easily to billions of data points, supports fault tolerance, and enables iterative optimization across workers. This makes Spark's GLM highly efficient and far more practical for large-scale machine learning tasks than its R or Python equivalents.

a Decision Tree Regressor

Scenario:
Suppose you're analyzing e-commerce data to predict customer spending based on features like age, location, number of site visits, time spent on pages, and clickstream patterns. The relationship between these features and spending is non-linear and includes complex interactions—for example, younger customers from certain regions may spend more only when browsing during specific times of day.

Why Decision Tree GLM Performs Better:
In this case, using a Decision Tree Regressor (such as sklearn.tree.DecisionTreeRegressor in Python) provides better performance than a linear model (GLM) from base R's glm() or sklearn.linear_model.LinearRegression. Linear models assume additive, linear relationships and struggle with sharp thresholds or interaction effects unless explicitly modeled, which requires manual feature engineering.

Decision Trees automatically capture non-linearities and interactions by splitting the data at optimal thresholds. They are non-parametric and do not assume a specific functional form, making them better suited for data where the underlying relationships are complex and unknown.

Result:
The Decision Tree GLM fits the data more naturally, requires less preprocessing, and yields more accurate predictions with minimal tuning—outperforming both base R's and scikit-learn's linear model equivalents in this non-linear setting.