AI and Data Science Journal
Online ISSN: 3078-7394
DOI: https://doi.org/10.61784/adsj3035

CLOUD-NATIVE DISTRIBUTED SYSTEMS FOR REAL-TIME PAYMENT INTELLIGENCE

BingJie Zi
Northeastern University, Boston 02115, Massachusetts, USA.

Abstract: Cloud-native payment platforms increasingly route every transaction through an online fraud-scoring microservice whose response time directly affects the end-to-end approval latency seen by merchants and cardholders. In this work we present a controlled empirical study of seven CPU-friendly classifiers for real-time payment fraud detection, evaluated on the public credit-card transaction dataset released by the ULB Machine Learning Group and Worldline. Models are trained on the first 60% of the transaction stream in chronological order and evaluated on the subsequent 40%, with thresholds tuned on an intermediate validation window under a precision-floor service-level constraint. For each detector we jointly report ranking quality (ROC-AUC, PR-AUC), operating-point precision and recall, single-row inference latency at the median, 95th and 99th percentiles, batch throughput, and serialized model footprint. We find that a regularized logistic-regression baseline attains a PR-AUC of 0.74 with a sub-millisecond p99 single-row latency and a 2 KB on-disk footprint, while a 50-tree random forest achieves the highest PR-AUC (0.81) at roughly an order of magnitude higher p99 latency and several hundred times the footprint. We further examine a sample-weighted incremental update of an online stochastic-gradient model over the validation stream and observe that, on this benchmark, simple incremental updates can shift the decision boundary in a way that increases false-positive load without improving recall, motivating more careful update strategies in production cloud-native pipelines. The implementation package can be released to support reproducibility.
Keywords: Real-time payment intelligence; Fraud detection; Cloud-native systems; Microservices; Online learning; latency; Class imbalance; Reproducibility

1 INTRODUCTION

Modern payment platforms are increasingly built as cloud-native distributed systems composed of small, independently deployable services that communicate through well-defined network interfaces [1]. A typical authorization path traverses a payment gateway, an API gateway, one or more risk-scoring services, a decision service, and finally the authorization and clearing layer. In such an architecture the fraud-detection model is invoked synchronously on every transaction, and its response time directly contributes to the end-to-end latency budget that the platform offers to merchants. Empirical analyses of large web services have long argued that controlling the tail of the latency distribution—rather than the mean—is what determines the responsiveness perceived by clients at scale [2]. For a payment scorer this translates into hard constraints on the 95th- and 99th-percentile inference latency per single transaction.

At the same time, fraud detection is an inherently difficult learning problem. The class distribution is heavily skewed, with typical fraud rates well below one percent of the transaction volume, and the underlying generative process is non-stationary: customer behavior drifts, fraudsters change strategies, and labels arrive with significant verification latency [3-4]. Practical fraud-detection pipelines must therefore balance ranking quality, calibration of the operating point, robustness to drift, and the engineering cost of deployment and re-training [5-7].

A large body of work has proposed sophisticated detectors—deep neural networks, graph models, ensemble cascades, and streaming frameworks built on big-data engines such as Apache Spark [6]. In this paper we take a deliberately narrower view. We ask: among the family of detectors that can be served comfortably within a single, lightweight inference container, how does the achievable detection quality trade off against the per-request latency and the on-disk footprint that the platform incurs at each invocation? Such a comparison is the question an SRE or platform engineer faces when provisioning a fraud-scoring microservice.

The contribution of this paper is fourfold. First, we describe a controlled, reproducible benchmark that combines a chronological train/validation/test split of the public ULB credit-card fraud dataset with a precision-floor operating-point selection rule that mirrors an alert-budget service-level objective [3-4]. Second, we evaluate seven CPU-friendly detectors—including linear, tree, ensemble, histogram-gradient-boosting and stochastic-gradient online variants—jointly along five axes: ROC-AUC, PR-AUC, operating-point F1, single-row p50/p95/p99 inference latency, batch throughput, and serialized model size. Third, we report the effect of a single sample-weighted incremental update of an online stochastic-gradient model over the validation stream, and analyze why such updates can degrade rather than improve precision at the chosen operating point. Fourth, we release the full reproducibility package, including data download script, deterministic training pipeline, and figure generation.

The remainder of the paper is organized as follows. Section 2 positions the work with respect to prior fraud-detection benchmarks, streaming detection frameworks, and the cloud-native systems literature. Section 3 formalizes the problem and the evaluation protocol. Section 4 describes the experimental setup. Section 5 presents and discusses the results. Section 6 concludes with practical takeaways and directions for future extension.

2 RELATED WORK

Credit-card fraud detection has been extensively studied; comprehensive surveys cover both algorithmic and operational aspects [5]. The public European cardholder transaction dataset that we use was assembled by Dal Pozzolo et al. and has been a standard benchmark since its release [3-4]. The same group of authors has examined the practical operating conditions of fraud-detection systems, including how class imbalance, concept drift, and delayed labels jointly shape performance assessment, and has proposed realistic protocols for evaluation [4]. Carcillo et al. extended these ideas to a distributed streaming setting [6], building a Spark-based framework (SCARFF) that ingests transaction streams and re-trains classifiers as labels arrive. Our study is intentionally narrower: we focus on the inference-side cost of a single, lightweight detector in a cloud-native microservice. Concept drift is reflected in the temporal evaluation protocol, but drift adaptation is not the central focus of this study.

On the systems side, the cloud-native architectural style emphasizes small, independently deployable services and is associated with operational benefits such as agility, isolated scaling, and resilience [1]. The flip side is operational complexity, in particular the difficulty of keeping the latency distribution tight when many small services compose into a single user-visible request [2]. The implication for machine-learning components is that detectors which are accurate in offline benchmarks but expensive at inference time can become a bottleneck for the entire pipeline.

Methodologically, our evaluation follows the recommendation that, for heavily skewed problems, precision–recall curves provide a more informative picture of performance than receiver-operating-characteristic curves alone [8-9]. We report both, together with operating-point metrics tied to a configurable precision floor. Our models span the canonical CPU-friendly families: regularized logistic regression and Gaussian naïve Bayes as linear baselines; a depth-limited decision tree as a fast non-linear model; a small random forest [10]; the histogram-based gradient-boosting variant available in scikit-learn [11], which is in spirit similar to LightGBM [12]; and a stochastic-gradient online classifier in the spirit of Bottou's large-scale SGD work [13].

3 METHOD

3.1 Problem Setting

We treat fraud detection as a binary classification task in which each transaction $x \in \mathbb{R}^d$ must be scored in real time, producing a score $s(x) \in \mathbb{R}$ that is then thresholded by a decision service to produce a binary action. We model the scoring component as a stateless function deployed behind an API gateway in a cloud-native pipeline, as sketched in Figure 1. The per-request latency budget allocated to the model is a fraction of the end-to-end approval-time budget, which in production payment platforms often requires low-latency responses within tight operational budgets. Within this budget the model must produce a score and the decision service must apply business rules, route the transaction, and communicate with the authorization network.

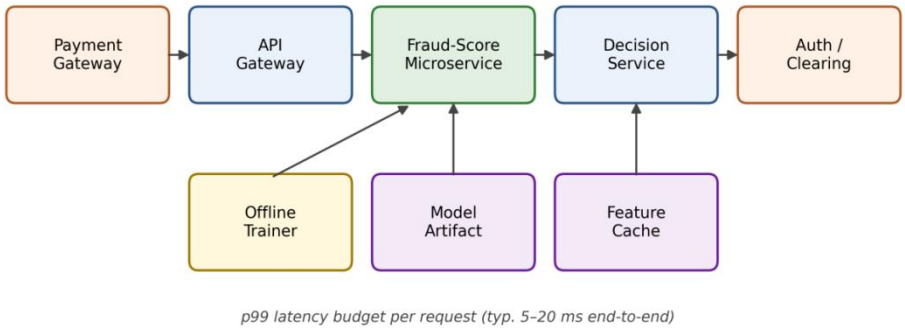

Figure 1 Inference Path of a Real-Time Fraud Detector inside a Cloud-Native Payment Platform

Note: The fraud-scoring microservice loads a model artifact produced by an offline trainer and reads pre-computed features from a feature cache. The per-request latency budget allocated to scoring is a fraction of the platform's end-to-end approval latency.

3.2 Dataset and Chronological Split

We use the public European cardholder dataset released by the ULB Machine Learning Group and Worldline [3-4]. The dataset contains 284 807 transactions collected over a 48-hour window, of which 492 are labeled as frauds (a base rate of 0.173%). Each transaction is described by 30 numerical features: a Time field that records the seconds elapsed since the first transaction in the dataset, an Amount field, and 28 features V1–V28 that are linear combinations produced by a principal-component analysis to protect commercial confidentiality. The label is binary.

Because production deployment is intrinsically temporal, we partition the dataset chronologically. The first 60% of transactions in time order form the training window, the next 20% the validation window, and the final 20% the test window. The resulting fraud rates differ across windows—0.211% in training, 0.100% in validation, and 0.132% in test—which is itself a mild form of distribution shift and is preserved deliberately so that the evaluation reflects realistic deployment conditions.

3.3 Models

We compare seven CPU-friendly detectors that can be served by a single inference replica without GPU acceleration: (i) a regularized logistic regression with balanced class weights and L2 regularization; (ii) a Gaussian naïve Bayes classifier; (iii) a single decision tree with maximum depth eight and balanced class weights; (iv) a random forest with 50 trees and maximum depth ten; (v) a histogram-based gradient-boosting classifier with 100 iterations, depth six, and learning rate 0.1; (vi) an online stochastic-gradient logistic regression with logistic loss, balanced class weights, and L2 regularization; and (vii) the same online stochastic-gradient model after one pass of sample-weighted incremental updates over the validation stream. All models are implemented using scikit-learn and are wrapped in a standardization pipeline where appropriate [11]. Hyperparameters were selected to represent practical lightweight deployment configurations rather than exhaustive offline optimization.

3.4 Evaluation Protocol

For ranking quality we report the area under the ROC curve and the area under the precision–recall curve on the test window. The precision–recall area is the primary ranking metric in our analysis because of the dataset's extreme skew [9]. For each model we additionally select a single operating threshold on the validation window using a precision-floor rule: the threshold is the most permissive value whose validation precision is at least 10%, with a tie-break that maximises validation recall under that precision floor. If no threshold satisfies the precision floor, the score that maximises validation F1 is used as a fallback. This rule corresponds to an SLA-driven calibration in which the analyst team has a fixed alert budget and prefers higher recall as long as the manual-review precision is reasonable.
To assess deployment cost we benchmark single-row inference latency on a fixed sample of 500 test transactions, drawn deterministically with a seeded random generator. Each transaction is scored individually in a Python loop after a warm-up of 50 single-row predictions. The procedure is repeated three times, producing 1 500 latency samples per model from which we report the median (p50), 95th-percentile (p95) and 99th-percentile (p99) latencies, all in microseconds. We additionally measure the batch throughput by scoring the full benchmark batch in a single call and dividing by the wall-clock time. Finally we serialize each fitted estimator with the standard Python pickle protocol and report the resulting file size in kilobytes as a proxy for the in-memory footprint of the model artifact.

3.5 Incremental Update of the Online Model

To probe the value of online updates in a cloud-native deployment we apply one additional pass of incremental learning to the stochastic-gradient model. After initial training on the training window, the model is updated in mini-batches of 256 transactions sweeping through the validation window in chronological order. Class weights are converted to explicit per-sample weights computed once from the prior of the validation window so that the partial_fit call can proceed even when some mini-batches contain no positive examples. The standardization statistics inside the pipeline are updated with the same mini-batch stream. The resulting model is then re-evaluated on the (still untouched) test window using the same precision-floor protocol.

4 EXPERIMENTS

4.1 Implementation

The pipeline is implemented in approximately 350 lines of Python using pandas, NumPy and scikit-learn [11]. All experiments are run on a single CPU core inside a Linux container with 32 GB of memory and no GPU; the global random seed is fixed at 42 to ensure reproducibility. Total wall-clock training time across all seven models is below 40 seconds. The dataset is obtained from a publicly accessible repository of the original benchmark and cached locally.

4.2 Procedure

For each model we (i) fit the estimator on the training window, (ii) score the validation window and the test window, (iii) select an operating threshold on the validation window using the precision-floor rule of Section 3.4, (iv) compute classification metrics on the test window at that threshold, (v) measure single-row latency and batch throughput on a fixed 500-transaction subset of the test window, and (vi) record the serialized model size. The incremental update of the online model is applied after step (i) using the validation window as the update stream, after which steps (ii)–(vi) are repeated. To ensure reproducibility, all reported results in Section 5 are produced under a fixed random seed and a deterministic pipeline.

5 RESULTS AND DISCUSSION

5.1 Ranking Quality

Figure 2 reports the ROC and precision–recall curves for all models on the test window. In ROC space the logistic regression, random forest and stochastic-gradient detectors are tightly clustered with areas of 0.97–0.98, while the histogram gradient-boosting model lags at 0.73 and the small decision tree at 0.86. In precision–recall space the picture

changes substantially: the random forest reaches an average precision of 0.81, the logistic regression 0.74, the decision tree 0.50, the histogram gradient-boosting 0.34, and both stochastic-gradient variants stay below 0.07. The reordering between the two views is a direct illustration of the well-known observation that strong ROC performance can coexist with weak precision at the top of the score distribution when the class distribution is highly skewed [9]. The online stochastic-gradient model is a particularly clear example: its overall ranking of frauds versus genuine transactions is competitive, but the absolute scores it assigns to the small fraction of true frauds are not separated cleanly from a much larger background of borderline genuine transactions.

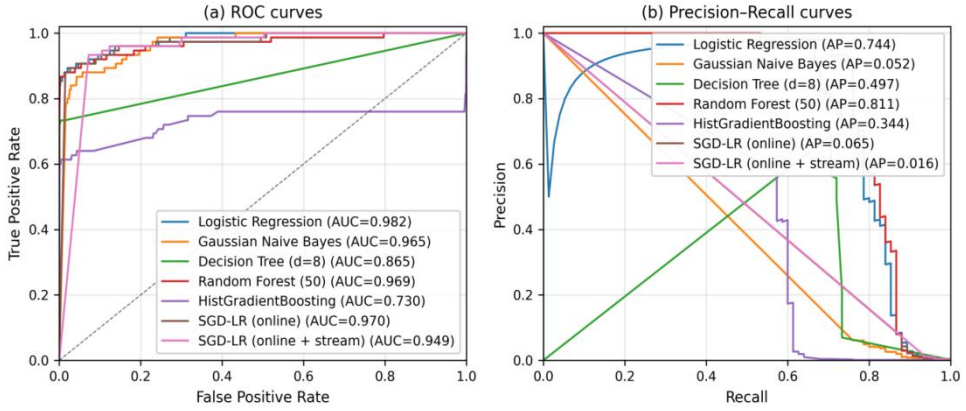

Figure 2 Ranking Quality on the Chronological Test Window

Note: (a) ROC curves; (b) precision–recall curves. Areas are annotated in the legend. The reordering of models between ROC space and PR space is consistent with [9].

5.2 Operating-Point Performance

Table 1 summarizes the main results, including the chosen operating point. Figure 3 visualizes the corresponding confusion matrices for four representative detectors. The logistic regression catches 64 of 75 test frauds while raising 225 false alerts and obtains an F1 of 0.352. The random forest catches 65 frauds at the cost of 313 false alerts (F1 = 0.287). The histogram gradient-boosting model is more conservative: it raises only 82 false alerts but misses 30 frauds, yielding the highest single-model F1 in this study (0.446) at substantially lower recall (0.600). The online stochastic-gradient model after the incremental update moves toward a high-recall, low-precision operating regime, catching 70 of 75 frauds but raising 4 110 false alerts, which corresponds to a precision of 1.7% and an F1 of 0.033.

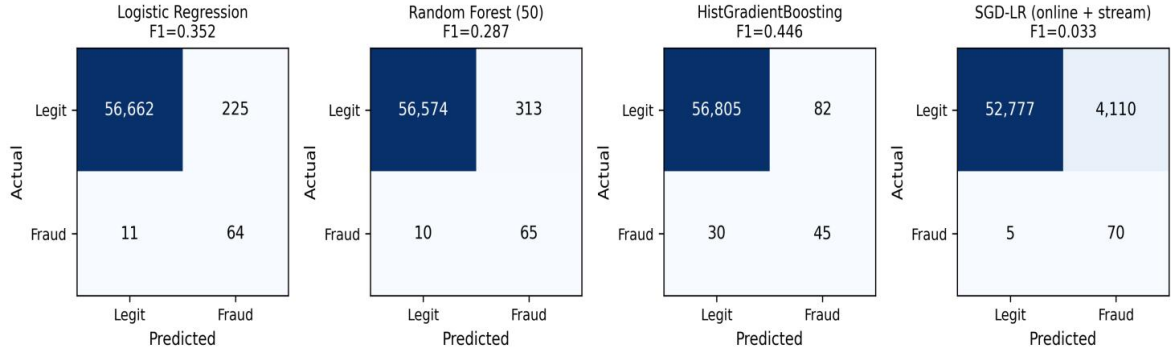

Figure 3 Confusion Matrices at the Precision-Floor Operating Point for Four Representative Detectors

Note: Cell counts illustrate how the small per-model differences in operating threshold translate into substantially different false-positive workloads.

Table 1 Main Results on the Chronological Test Window (Precision-Floor Operating Point Selected on the Validation Window)

Model	Paradigm	ROC-AUC	PR-AUC	F1	p99 latency (μs)	Throughput (txn/s)	Size (KB)
Logistic Regression	batch	0.982	0.7439	0.3516	340.2	1433991	2.0
Gaussian Naive Bayes	batch	0.9653	0.052	0.1129	374.5	892987	2.1
Decision Tree (d=8)	batch	0.8647	0.497	0.3293	106.8	4883384	14.6
Random Forest (50)	batch	0.9689	0.8115	0.287	3559.2	119385	779.9
Hist. Gradient Boosting	batch	0.7303	0.3439	0.4455	226.3	1065146	95.7
SGD-LR (online)	online	0.9696	0.0648	0.1328	264.5	1517708	2.1

Model	Paradigm	ROC-AUC	PR-AUC	F1	p99 latency (µs)	Throughput (txn/s)	Size (KB)
SGD-LR (online + stream)	online+upd.	0.9486	0.0161	0.0329	266.3	1512008	2.1

Note: All latency figures are single-row inference times measured on one CPU thread; throughput is batch throughput on 500 transactions.

5.3 Latency, Throughput and Footprint

Figure 4 places the seven detectors on a single trade-off plot whose horizontal axis is the p99 single-row latency on a log scale and whose vertical axis is the test-set PR-AUC. The marker area is proportional to the square root of the serialized model size. The logistic regression lies near a favorable region of the Pareto frontier: it combines a PR-AUC of 0.74 with a sub-millisecond p99 latency and a 2 KB on-disk footprint. The depth-eight decision tree achieves the lowest p99 latency in the study at a PR-AUC of 0.50. The histogram gradient-boosting model offers similarly low latency but with lower PR-AUC (0.34) under our default hyperparameters. The random forest reaches the highest PR-AUC (0.81), but its p99 latency is roughly an order of magnitude higher than that of the logistic regression and its serialized footprint of about 780 KB is several hundred times larger. Exact latency, throughput and footprint values are reported in Table 1.

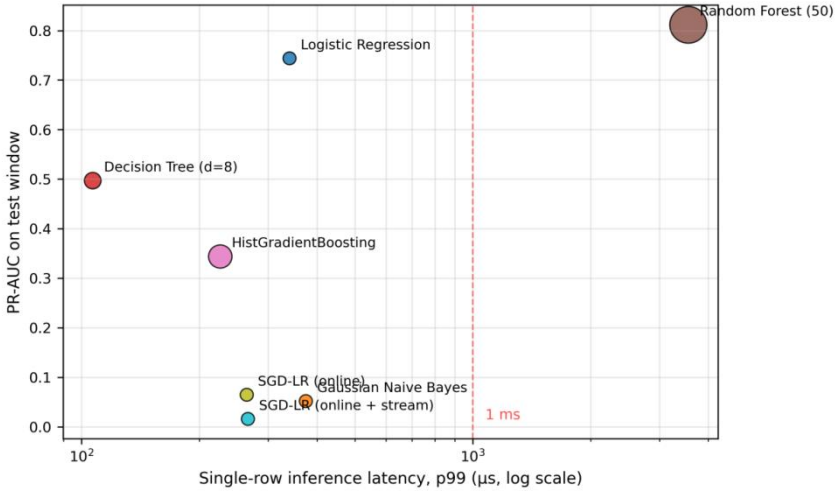

Figure 4 Latency / PR-AUC Trade-off on the Test Window

Note: The marker area is proportional to the square root of the serialized model footprint. The red dashed line marks a 1 ms p99 budget, beyond which a per-request scoring service starts to consume a sizable fraction of a typical end-to-end approval-time budget.

Throughput numbers tell a complementary story. In batch mode all linear and tree-ensemble models score on the order of one million transactions per second per core, while the random forest is the only model whose throughput sits clearly below this level (approximately 10^5 transactions per second, an order of magnitude lower). At sub-millisecond p99 budgets the random forest therefore not only spends more wall-clock time per request but also leaves less headroom for any other work that the scoring service has to perform inside its container.

5.4 Effect of the Incremental Update

Comparing the two variants of the stochastic-gradient model in Table 1 shows that the additional pass of incremental updates over the validation window decreased the test PR-AUC from 0.065 to 0.016 and the operating-point F1 from 0.133 to 0.033, even though it increased the operating-point recall slightly (from 0.88 to 0.93). The update step was deliberately conservative: it used the existing inverse-prior sample weighting, kept the same learning-rate schedule, and processed the validation stream once in chronological order. One likely contributing factor is that the validation window has a lower fraud rate than the training window and contains a different mix of fraudulent transactions. Applying sample-weighted updates from this stream pushed the linear decision boundary further away from the fraud cluster that the training data had isolated, but did so by moving the score distribution as a whole rather than by improving the separation between classes.

This result highlights that incremental adaptation can be sensitive to class-prior shifts and update protocols, especially when the validation stream differs from the original training window. The intent is not to argue that online updates do not help in general but to illustrate that, in a production cloud-native fraud detector, a simple incremental update of an already-fitted model does not automatically translate into improved deployment performance. More careful approaches—periodic re-training on rolling windows, ensemble blending of a frozen base model with an incremental delta, or active-learning-style query selection [7]—are well-established in the streaming fraud-detection literature and are clearly preferable to the simplest update strategy explored here.

5.5 Practical Implications

Three practical observations follow from the experiments. First, a well-regularized logistic regression offers the most favorable latency–accuracy trade-off among the evaluated models for cloud-native fraud detection: it occupies an attractive corner of the latency–quality plane on this benchmark and is straightforward to deploy, version and roll back. Second, moving to a small random forest is justified only when the additional 0.07 of PR-AUC clearly outweighs roughly an order-of-magnitude increase in p99 latency and a comparable increase in serialized footprint; under strict sub-millisecond latency constraints the random forest may be less suitable. Third, the choice of operating-point selection rule matters as much as the choice of model, since the same ranking can be turned into very different alert-budget profiles. Reporting both ranking metrics and operating-point metrics, as recommended in [9], is therefore important for any apples-to-apples comparison of fraud detectors.

6 CONCLUSION

We presented a controlled, reproducible study of seven CPU-friendly classifiers for real-time fraud detection in a cloud-native payment pipeline. By combining a chronological train/validation/test split of a public transaction dataset with a precision-floor operating-point selection rule, we obtained a benchmark that simultaneously captures ranking quality, operating-point performance, single-row latency at multiple tail percentiles, batch throughput and model footprint. The most favorable latency–accuracy trade-off on this benchmark is achieved by a regularized logistic regression: it attains a PR-AUC of 0.74 with a sub-millisecond p99 single-row latency and a footprint of about 2 KB. A small random forest achieves the highest PR-AUC (0.81) but at roughly an order of magnitude higher p99 latency and several hundred times the footprint. A simple incremental update of an online stochastic-gradient model over the validation stream did not improve performance on this benchmark, highlighting that incremental adaptation can be sensitive to class-prior shifts and update protocols in production deployments.

The study prioritizes a controlled and reproducible comparison using a widely adopted public transaction benchmark and lightweight CPU-oriented detectors. Natural extensions include evaluating the same protocol on additional public transaction streams and exploring more sophisticated incremental update strategies that combine periodic re-training with sliding-window calibration.

COMPETING INTERESTS

The authors have no relevant financial or non-financial interests to disclose.

REFERENCES

[1] Jamshidi P, Pahl C, Mendonça N C, et al. Microservices: The journey so far and challenges ahead. IEEE Software, 2018(3): 24–35.
[2] Dean J, Barroso L A. The tail at scale. Communications of the ACM, 2013(2): 74–80.
[3] Dal Pozzolo A, Caelen O, Johnson R A, et al. Calibrating probability with undersampling for unbalanced classification. Proc. IEEE Symposium Series on Computational Intelligence (SSCI), 2015: 159–166.
[4] Dal Pozzolo A, Boracchi G, Caelen O, et al. Credit card fraud detection: A realistic modeling and a novel learning strategy. IEEE Transactions on Neural Networks and Learning Systems, 2018(8): 3784–3797.
[5] Dal Pozzolo A, Caelen O, Le Borgne Y A, et al. Learned lessons in credit card fraud detection from a practitioner perspective. Expert Systems with Applications, 2014(10): 4915–4928.
[6] Carcillo F, Dal Pozzolo A, Le Borgne Y A, et al. SCARFF: A scalable framework for streaming credit card fraud detection with Spark. Information Fusion, 2018: 182–194.
[7] Carcillo F, Le Borgne Y A, Caelen O, et al. Streaming active learning strategies for real-life credit card fraud detection: assessment and visualization. International Journal of Data Science and Analytics, 2018(4): 285–300.
[8] Bhattacharyya S, Jha S, Tharakunnel K, et al. Data mining for credit card fraud: A comparative study. Decision Support Systems, 2011(3): 602–613.
[9] Davis J, Goadrich M. The relationship between Precision-Recall and ROC curves. Proc. 23rd International Conference on Machine Learning (ICML), 2006: 233–240.
[10] Breiman L. Random forests. Machine Learning, 2001(1): 5–32.
[11] Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 2011: 2825–2830.
[12] Ke G, Meng Q, Finley T, et al. LightGBM: A highly efficient gradient boosting decision tree. Advances in Neural Information Processing Systems (NeurIPS), 2017: 3146–3154.
[13] Bottou L. Large-scale machine learning with stochastic gradient descent. Proc. COMPSTAT 2010, 2010: 177–186.