# Large-Scale Personalized Categorization of Financial Transactions

*Christopher Lesner, Alexander Ran, Wei Wang, Marko Rukonic*

■ *A major part of financial accounting involves organizing business transactions using a customizable filing system that accountants call a "chart of accounts." This task must be carried out for every financial transaction, and hence automation is of significant value to the users of accounting software. In this article we present a large-scale recommendation system used by millions of small businesses in the USA, UK, Australia, Canada, India, and France to organize billions of financial transactions each year. The system uses machine learning to combine fragments of information from millions of users in a manner that allows us to accurately recommend chart-of-accounts categories even when users have created their own or named them using abbreviations or in foreign languages. Transactions are handled even if a given user has never categorized a transaction like that before. The development of such a system and testing it at scale over billions of transactions is a first in the financial industry.*

In standard classification, the task is to learn a function that can map each input object to a class. The input objects are represented by sets of values called *features*. The classes, also called labels, are represented by a set of symbols or nominal values. It is an important assumption of standard classification that classified objects can be represented by *n*-dimensional vectors of real-valued features that are independent and identically distributed over the features, while classes, it is assumed, can be represented by a single nominal attribute from a finite, unordered set. Given a sufficiently large set of input space objects with corresponding labels, such standard classification problems can be efficiently solved with off-the-shelf classification algorithms. However, in many applications, these assumptions of standard classification do not hold. Here, we discuss a classification problem in which classes cannot be represented by nominal attributes only; classes have user-specific scope; classes are evolving in the very process where classification is used by a large community of users; and the domain of the most important features comprises nominal sets having tens or hundreds of thousand elements.

Although this situation is not uncommon, relatively little has been published about such personalized classification problems. In this article, we share our experience with personalized classification of financial transactions for automating small business accounting. The insights and conclusions of our work should be readily applicable in domains characterized by a large community of users evolving their personal categorization taxonomies in a way that is immediately relevant for their specific context while having varying degrees of similarity to how other users organize their information. Such domains include filing of financial transactions for personal and business accounting, organization of family photographs, organization of personal collections of movies, cooking recipes, website bookmarks, and others.

In this article, we will first explain the application domain and the value the automation of small business accounting has for millions of small businesses around the world. We then explain why standard classification methods fall short for the purpose of financial transactions classification, and present a framework we developed to solve personalized classification problems — a *confidence-based ensemble of association strength rankers* (CEASR). Next, we discuss what was necessary to build the working system that has been serving millions of Intuit QuickBooks customers for the last several years. We share some performance metrics and our experience with large-scale–model training and deployment. Here we emphasize the importance of different data representations — one suitable for model training and another for model deployment. We also discuss what sorts of faults can occur during model builds, how we detect these faults, and how our system uses checkpoints for recovery from faults. We then cover some practical aspects of dealing with soft real-time latency deadlines, and selecting and optimizing servers for model builds versus model deployment. We conclude with a discussion of user impact, benefits, and what we learned.

## Accounting Automation with Personalized Classification of Financial Transactions

Financial accounting organizes business transactions using a customizable filing system accountants call a *chart of accounts* (CoA). To keep business books organized, every transaction must be filed using the CoA — even small purchases and payments — so this is a tedious chore for millions of accounting software users. Assuming it takes about three seconds for a typical financial transaction, last year the users of Intuit's accounting software would have spent well over a thousand man-years on this task, if not for the assistance provided by automation.

Assigning correct categories to financial transactions is important because errors on this task can lead to incorrect financial statements, increased audit risk, tax, and other regulatory penalties, misinformed financial decisions, and displeased creditors and investors. For these reasons, accurate financial transaction filing is of significant economic value for everyone involved — business owners, their accountants, vendors of accounting software, and others.

In this article, we present a large-scale recommendation system used by millions of small businesses in the USA, UK, Australia, Canada, India, and France to organize billions of financial transactions each year. The system uses machine learning to combine fragments of information from millions of users in a manner that allows us to accurately recommend CoA categories even when users have created their own or named them using abbreviations or in foreign languages, and transactions are handled even if a given user has never categorized a transaction like that before. The development of such a system and testing it at scale over billions of transactions is a first in the financial industry.

## Domain Information Structure

A simplified model of accounting of payments is shown in figure 1. Each company uses one or more financial institutions that offer financial accounts that facilitate transactions such as receiving money from customers, paying wages to employees, and paying bills to suppliers and service providers. The records of financial accounts can be electronically downloaded and each financial transaction must then be given an accounting interpretation using a CoA, which is a customizable collection of accounting categories. Our focus in this article is on a system that learns to recommend the most suitable CoA account for each down-loaded transaction.

Financial accounts track how much money changed hands on a given date with a certain counterparty, but unlike an invoice or a receipt, the transaction records from a financial account typically do not have information about the items purchased or the services involved. Financial transaction records generally only include: a transaction description (may refer to counterparty); a financial institution that recorded the transaction; a financial account description; a date of the transaction; and a money amount.

Knowing the counterparty merchant, vendor, or service provider can help assign a transaction to the correct CoA category. However, it is frequently not possible to know this just from transaction descriptions provided by users' financial institutions. At best, we are able to infer whether two transactions refer to the same counterparty in their descriptions, through a form of probabilistic coreference resolution (Clark and Manning 2015). The details of our approach for coreference resolution are outside the scope of this article. What is important, however, is that real-world attributes of the counterparty such as the name, description, business domain, and the like, all of which can be helpful for inferring the meaning

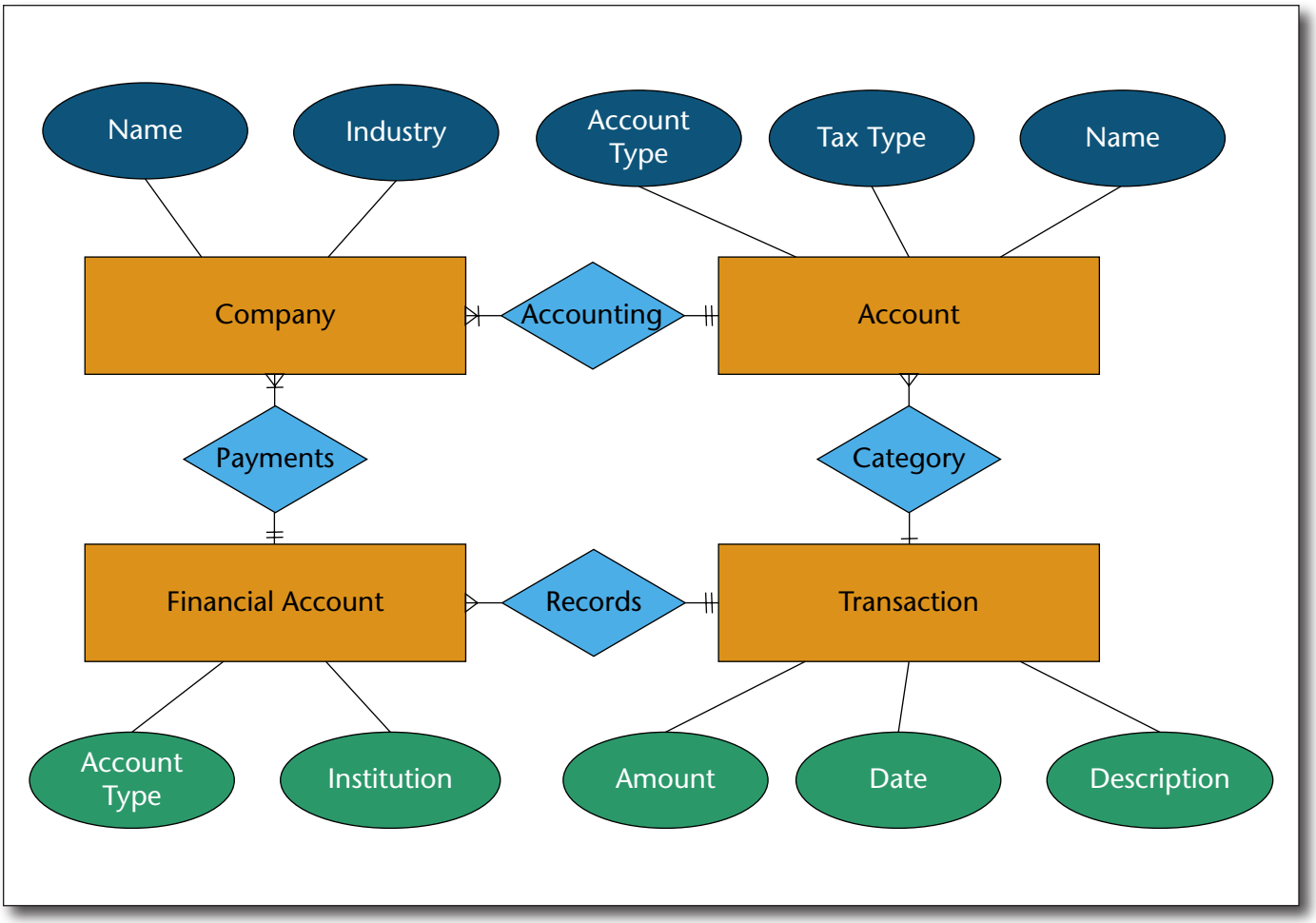


*Figure 1. Accounting of Payments.*

Simplified model.

of the transaction in the accounting domain, are not necessary to build our system.

The main transaction attribute, the transaction description, is categorical, nominal with cardinality in the order of $10^8$. We estimate that there are only about $10^7$ distinct counterparties and the extra order of magnitude comes from the imperfect coreference resolution. This clearly violates the typical assumptions of algorithms designed for standard classification. For example, a typical approach to represent nominal values is one-hot encoding. Each nominal value from a set of $n$ values is represented as an orthonormal vector in $n$-dimensional space. Instead of a single categorical feature, we have $n$ features only one of which has the value of one, and the rest are zeros. In our case, just representing the transactions counterparty would require hundreds of thousands features.

To facilitate business insights from accounting reports, small business accounting systems enable companies to customize their CoA categories and, in practice, semantic information about the CoA of any given company is either unreliable or unavailable. For example, CoA account attributes such as its *type* (indicating whether the account represents income, expenses, cost of goods sold, fixed assets) or *tax type* (for example, using IRS schedule C) are only reliable if the small business uses reports and analysis that depend on the correct setup of these attributes.

Many small businesses treat their CoA as a set of folders for organizing related transactions and the CoA account name is the only attribute that is important for them. Across different companies, the same account names may have different meanings, and different account names may have the same or similar meanings.

Furthermore, companies organize their transactions using different levels of granularity. Transactions pertaining to internet service, cellular phones, water, and gas services may all be filed in the same utility's CoA account; or tracked using individual vendor accounts such as Comcast, Verizon, Sprint, and PG&E; or tracked as communication services versus water service versus heating service. As a result, much like counterparty identities, accounting categories are best thought of as nominal attributes unique for

a particular company. Instead of a finite, moderately sized set of classes with no structure, we have a complex, large set of objects.

## Baseline for Automation

The automation of transaction categorization can be thought of as learning a ranking function $\gamma(u, a, c) = r \in \Re$ that maps possible combinations of User, Company, Account, and Transaction, represented by their respective attributes (see figure 1) to a real number *r,* such that

$$\forall r_i, r_j : r_i = \gamma(u, a_i, c), r_j = \gamma(u, a_j, c)$$
$$r_i > r_j \Leftrightarrow P(a_i \mid u, c) > P(a_j \mid u, c) \quad (1)$$

There are special challenges for applying common solutions to a standard classification problem to this domain:

First, low-cardinality categorical attributes associated with companies, accounts, and transactions such as Industry, Tax Type, and Account Type, have low predictive power with respect to transaction to account assignment.

Second, low-dimensional representation of textual attributes such as Account Name, Transaction Description, and Company Name, perform worse on the categorization task than simple memorization of nominal associations.

Third, nominal representations of textual attributes have extremely high cardinality (tens or hundreds of millions).

Historic data may be represented as a set of tuples $\Gamma = (u, a, c, t) \in U \times A \times C \times (t_0, \text{now})$, where *u*, *a*, *c*, and *t* stand for the identities (references, nominal attributes) of the users/companies, accounting categories, transaction counterparties, and the time of the event, respectively. *U*, *A*, and *C* are the respective domains of identities for user, accounting categories, and transactions counterparties; and $(t_0, \text{now})$ is the time interval. The task is to learn a function $\gamma$: $U \times A \times C \rightarrow R$.

Notice that even this simple interpretation of the domain violates the assumptions of standard classification. For example, the number of classes is not fixed as users can and do define new accounting categories. The number of accounting categories actively used during one year by the entire community of QuickBooks[1] users is in the order of $10^8$.

Because accounting categories of different companies are represented by distinct nominal attributes and, therefore, historic data are a collection of associations between distinct nominal values, it seems like the only possibility is memorizing such associations between accounting categories and counterparties created by the user in the past using a ranking function that satisfies the requirements of equation 1. One such function uses the most popular category that the user has assigned to a given counterparty in the past $\gamma^p(u, a, c) = |\Gamma \cap (u, a, c,)$. Another such ranking function uses the timestamp of the last occurrence of the tuple (*u*, *a*, *c*) as the value of the ranking function $\gamma_t(u, a, c) = max(t: (u, a, c, t) \in \Gamma)$.

While either of these functions can predict accounting categories for counterparties present in the user's transaction history, no predictions can be made for transactions with new counterparties that constitute about fifty percent of all transactions. So even if future transactions with the same counterparty are always categorized correctly and new counterparty transactions are assigned to the most popular account, the mean accuracy of such a classifier would be at most around fifty percent, starting at close to zero percent for new users and slowly growing as the user accumulates a personal history of classified transactions. Learning such explicit mappings from the counterparty to the most likely accounting category independently for every user has been, and continues to be, the state of the practice today among many vendors of accounting software.

## Domain Graph for Coding Nominal Features

One may observe that counting the instances of transactions with a given counterparty in user's accounting categories and selecting the category with the maximum count (as specified by $\gamma_p$) is an approximation of maximum likelihood with the assumption that all categories are equally probable for a transaction with an unknown counterparty. The counts can be thought of as estimates of quantities proportional to conditional probabilities of the counterparty, given the user's account. What we need is a way to extend this procedure to counterparties with which the user has had no prior transactions.

Rather than interpreting the available data propositionally, when each tuple (*u*, *a*, *c*, *t*) is an independent fact, we interpret it as representing a graph of relationships among the users, accounting categories, and counterparties. Because identity of accounting categories can be unambiguously mapped to the identity of the user, we will only focus on the relational representation of accounting categories and counterparties. Let's assume that relationships between accounting CoA categories and counterparties are represented in the data by the similarity of their attributes and associations. This induces the following set of relationships: similarly named CoA categories; CoA categories with matching tax or account types; CoA categories with the same counterparties; counterparties assigned to the same CoA category; and counterparties assigned to related CoA categories.

For transactions with counterparties that a given user has categorized in the past, the strategies of using the most popular or the most recent category, for example, perform quite well. These strategies in essence represent the nominal counterparty attribute using a consistent scoring procedure over the set of accounts to be ranked, such that it preserves partial order of these accounts with respect to conditional

probability of the account given the counterparty. This coding of nominal attributes is often called *target coding*, as it substitutes a nominal attribute with an estimate proportional to the probability of observing the nominal attribute associated with the target class.

While direct observations only provide information about associations of a specific counterparty with a specific accounting category of a given user, by using the relational graph we can extend this idea to encompass other kinds of associations. In relational interpretation, each user account is represented by a graph induced by the relationships to counterparties and other accounts induced by an account's attributes and associations with transaction counterparties. Thus, each counterparty can be represented by a vector of scores. Each score is proportional to the conditional probability of the counterparty being associated with other accounts, related to the given account by the value of its attribute or its direct associations with other counterparties.

When a user has a transaction with a new counterparty, this counterparty can be represented by the strength of association with the set of accounts to be ranked for the classification task as well as by the strength of association with other entities related to the account, such as other accounts of the same type, similarly named accounts, and the counterparties present in the account. These can be derived from counterparty occurrence and co-occurrence statistics in the accounting categories of other users. This way we use strength of association between accounts and counterparties derived from the data of the entire population of users to estimate, for example, the strength of association between a new counterparty and user's accounting categories.

For another example, an account's type attribute relates a given accounting category to all other accounts of the same type. The fraction of transactions with a given counterparty that is associated with the accounts of the same type can serve as an estimate for a score that satisfies the requirements of the ranking function from equation 1.

*Account Name* is another attribute that can be used to relate accounting categories. Unlike the low-cardinality attribute such as *Account Type*, the Account Names have cardinality in the order of $10^5$ and only about half are shared between any two users. Rather than equality of Account Names, one can use a similarity measure to define the similarity-by-name relationship for accounting categories. Once such a relationship is defined, the process of scoring the counterparties by their strength of association with every accounting category can be the same as in other cases.

Finally, accounting categories become related by virtue of being associated with the same counterparty. One can think of this type of association as a second-order association. If we can score all pairs of counterparties proportionally to the probability of their co-occurrence in the same account, we can use such scores to score accounting categories with respect to the probability of association with a new counterparty, based on the current association of the accounting category with other counterparties. The types of relationships that exist in our domain model are shown in figure 2.

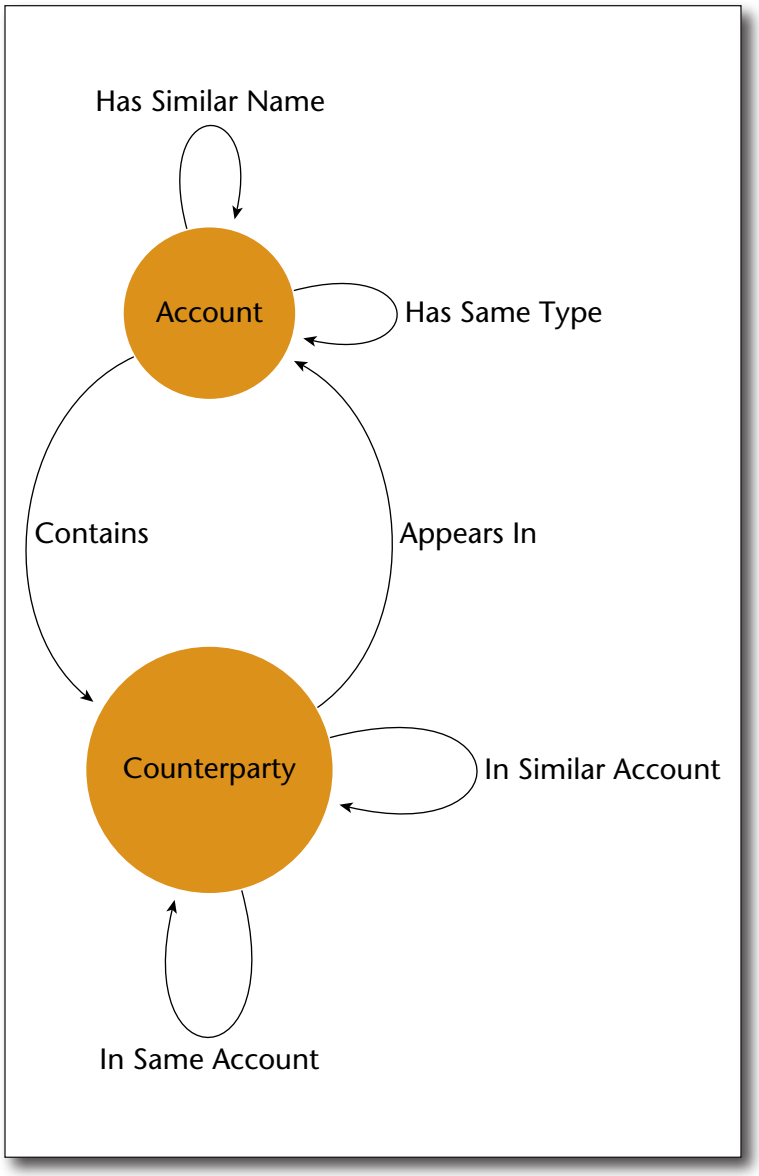


*Figure 2. Relational Model of Historic Data.*

## Event Counts for Association-Strength Scoring

Using event counts over sets of observations, representing categories, to estimate the probability of the event given the category can be effective when the number of observations is large and the events of interest are well distributed over the categories. When dealing with events defined by high-cardinality nominal attributes, as is the case in our domain,

there is a need to account for rare events. When counting counterparty distribution over accounts or counterparty distribution over accounts defined by type or name similarity, we add a fixed number of events to every category. This is known as *additive smoothing*, sometimes also called *Laplace smoothing* (Vechtomova 2009).

When prior probability of all categories to be ranked (typically a subset of accounting categories of a given company) can be estimated from data without knowledge of the counterparty of the new transaction, the posterior probability of the accounting category given the counterparty can be estimated as a mixture of two terms. The first term is estimation based on the counts associated with the counterparty. The second term is the prior for the category estimated from all training data $P(a|c) = \lambda(n_a)\, n_a/n + (1 - \lambda[n_a])P(a)$, where $n_a$ is the count of training-set transactions with counterparty $c$ and accounting category $a$, and $\lambda(n_a) \in (0, 1)$ is a monotonic function that increases from 0 for small $n_a$ and approaches 1 as $n_a$ increases. (See the work by Micci-Barreca (2001) for a discussion on choices.)

Similar problems have to be addressed when scoring accounting categories on second-order relationships such as, for example, counterparty-to-counterparty associations. Intuitively, the counterparty-to-counterparty association should be related to the likelihood that counterparty $c_i$ appears in the same accounting category $a$ as the counterparty $c_j$, given that the user has transactions with both counterparties. This can be estimated by counterparty co-occurrence statistics.

Care must be taken to address situations when the two counterparties have very different frequencies of occurrence ($Count(c_i) >> Count[c_j]$) as well as situations when one of the counterparties is rare. The first situation can be addressed by using a mean of fractions of cases when counterparty $c_i$ is present in the accounting category, given that the counterparty $c_j$ is present in the accounting category plus the inverse, as in $0.5(P(c_i \in a|\ c_j \in a) + P(c_j \in a|\ c_i \in a))$.

On the other hand, association strength will be grossly overestimated for rare counterparties that happen to co-occur once or twice. This problem has been observed by others using pointwise mutual information for estimating the strength of lexical association (see, for example, the work by Recchia and Nulty [2017]). We address this problem by adding a factor that scales down the estimation when one of the counts is comparable to $k$, and approaches 1 when both counts are large compared with $k$ (a small integer).

## Combining Weak Ranking Predictors

It is worth noting that coding counterparty representation by scoring association strength with each accounting category along multiple dimensions of association derived by a walk on the domain graph, produces multidimensional representation of the counterparty such that scores along each dimension satisfy the requirements of equation 1 (the ranking function equation), and thus can be used directly and independently as three weak ranking predictors:

Scores derived from direct associations between counterparties and accounting categories can be used to rank user's accounts when classifying a transaction with a counterparty known to the user.

Transactions with counterparties not known to the user can be classified by ranking a user's accounting categories based on the strength of association between the counterparty and the counterparties directly associated with user's accounts.

Transactions of users who have not classified any transactions before can be classified using scores derived from association strength between transaction counterparty and accounting categories of other users related by attribute equality or similarity to the user's accounting categories.

We have used this strategy and achieved good results as reported by Lesner et al. (2019), but this approach has several limitations. First, the performance of the ensemble is bounded by the performance of the individual base predictor applied; the combined power of all the available predictors is not used. Second, it is unclear how to integrate predictors that have the same applicability conditions such as, for example, multiple predictors derived by scoring a counterparty along different dimensions of account-to-account association.

To address these limitations, we developed a flexible approach to combine multiple ranking predictors where each predictor trains independently in parallel, which is necessary for scaling the solution to millions of users, hundreds of millions of accounting categories, and tens of millions of unique counterparties.

## Confidence Based Ensemble of Association Strength Rankers (CEASR)

Combining the output of multiple base predictors/classifiers to achieve a better performance than any of the base classifiers has been a goal of ensemble-learning research for more than 20 years (Freund and Schapire 1995; Efron and Tibshirani 1998; Grove and Schuurmans 1998), and remains an active area of interest (Montalbo and Festijo 2019).

A common recommendation of many researchers revolves around weighted voting for combining classification ranks from base classifiers. Weights can be determined through, for example, supervised learning on a separate dataset to maximize the performance of the ensemble (Li et al. 2014; Wang et al. 2015). For problem domains similar to ours that have a large number of classes, a relatively small number of examples per class, and a complex, evolving class structure, more-complex methods have been proposed as well (Melnik, Vardi, and Zhang 2004).

All approaches that we know of propose strategies that use classifier features measured or derived from classifier performance over a population of items. We have observed, however, that performance of different base predictors varies widely over the population of classified items. Thus, we saw an opportunity to develop an approach that is guided by the expected accuracy of the base classifier for the specific item being classified.

The core idea is for each base classifier to also train a separate model, called a *confidence model*, to estimate the probability that the top-ranked category recommended by the base classifier for the specific item being classified is correct. Such a confidence model can be trained with a representation of the item in some feature space using historical data for correct item class.

Because each of the base classifiers is a ranking classifier as defined in Equation 1, when classifying a counterparty $c$ for user $u$, having $n$ user-specific classes (accounting categories) $a_i$: $i \in (1..n)$, each of the base classifiers will produce a set of ranking predictions $r_i$: $i \in (1..n)$. While the highest-ranked class is the best answer the base classifier can give to the classification problem, the sequence of top $k$ ranks $r_i$: $i \in (1..k)$ provide a $k$-feature vector representation that effectively integrates information about the base classifier and the classified item predictive of the likelihood that the class selected by the base classifier is the correct class for the item.

For each base classifier, we train a confidence model $\lambda$: $R^k \rightarrow R$ that minimizes the mean-squared error with respect to the base classifier top-ranked class $a_{(1)}$ being the correct class for the item. Our algorithm then uses the sum of ranking scores produced by each base classifier, scaled by the estimated confidence of the base classifier for the classified transaction.

## Experimental Results

Two performance indicators directly impact how much work accounting software users must do to organize their financial transactions: *Accuracy of recommendations* — every inaccurate recommendation has to be manually corrected. *Accuracy of recommendation confidence* — sorting recommendations by how likely they are to need corrections makes the review process faster, because users can focus their attention on a small fraction of transactions that need it the most.

To track these performance indicators, we plot mean accuracy of recommendations against the fraction of all recommendations when sorted by descending confidence of prediction, as shown in figures 3, 4, 5, and 6.

Comparing figure 3 to figure 4, it is evident that in regions with more users and more classified transactions, the performance is better both in terms of absolute accuracy and in terms of our ability to sort transactions by expected accuracy.

Before CEASR, the mean accuracy of our recommendations was around seventy percent in the smaller region, and slightly above seventy percent in the larger region. Our ability to sort recommendations by expected accuracy was limited. In the smaller region, we could at best separate about a third of all transactions with expected accuracy above eighty percent. In the larger region, we could only identify about forty percent of transactions for which recommendations were ninety-percent accurate; however, with CEASR, we can separate seventy percent of transactions with a mean accuracy of category recommendations above ninety percent across different regions. The impact is also more pronounced in smaller user regions with less training data.

It is clear when comparing figures 3, 4, 5, and 6 that CEASR significantly improves performance, has greatest overall benefit in smaller regions, and shows consistent performance across all regions.

## How the Model Is Used

QuickBooks offers users the ability to connect their financial accounts (banks, credit unions, investments) to download transactions. What happens next is illustrated in figure 7. Upon download, each transaction undergoes analysis to understand what it represents (withdrawal or deposit, purchase or income, loan payment or disbursement, money transfer, fee) and who the transaction is with (counterparty). Next, our account likelihood-ranking model is applied and transactions are tentatively filed (autocategorized) with respect to each user's CoA. In the final step, users get an opportunity to accept or correct how their transactions have been filed, and their corrections are used to update the account likelihood-ranking model the next time it is built.

## How the Model Is Built

To keep production models fresh, we regularly rebuild them. This process has three main steps, as shown in figure 8.

### Data Extraction

Model builds start with extraction of just the table columns that pertain to financial account transactions and CoA accounts. From a data warehouse, these columns are transferred to Vertica[2] (figure 8, step A), where additional projections are added so that our model build data access patterns are sequential.

### Model Build

The model build (*computing the counterparty co-occurrence sparse matrix*, from here on, called the *coupling table*) is carried out in Vertica as controlled by a Python orchestration service. Once model tables are created, they are transferred from Vertica to PostgreSQL[3] (figure 8, step B) — in this step, our knowledge representation is switched from column store to row store (see "Knowledge Representation," below).

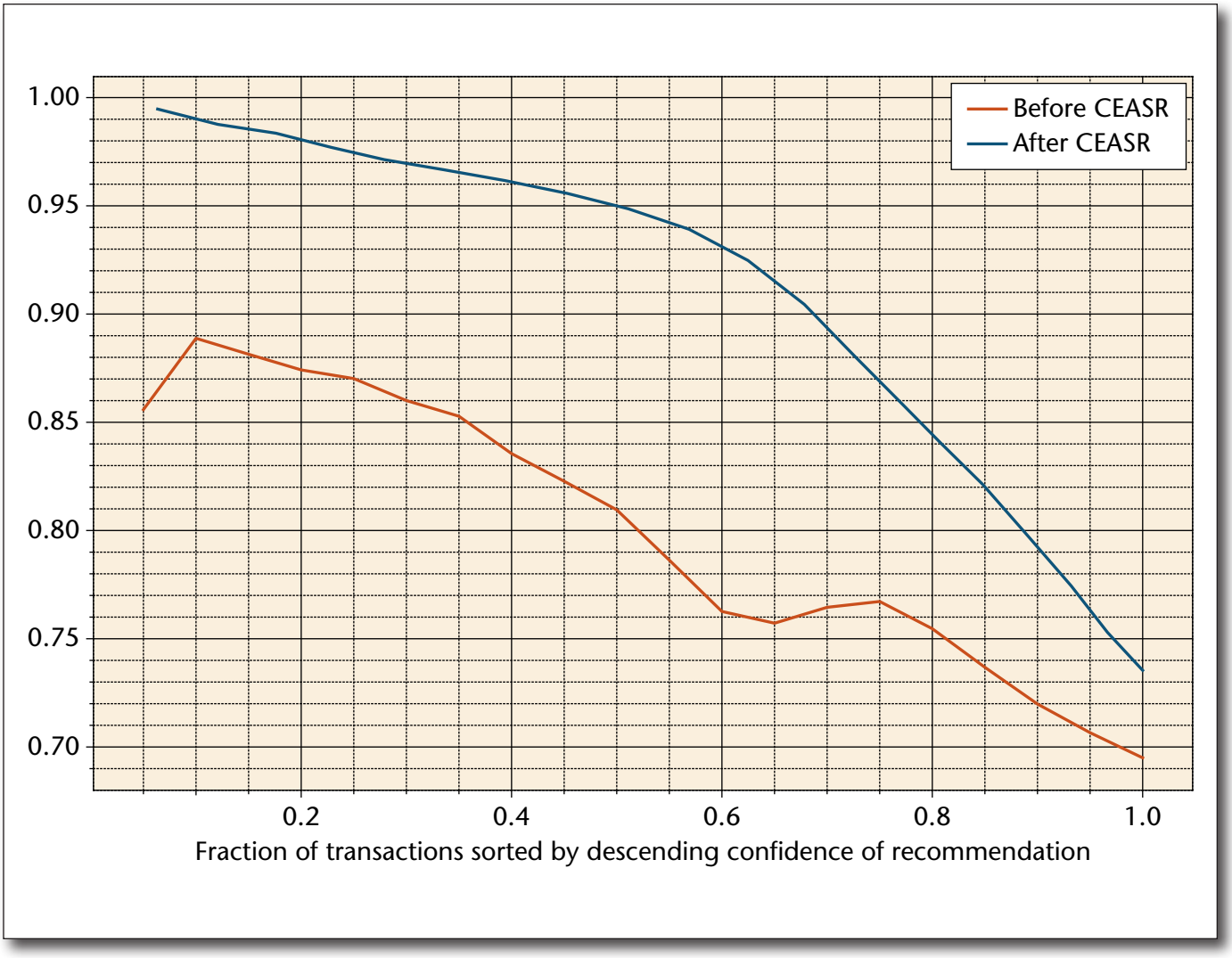


*Figure 3. Accuracy in a Smaller User Population Region.*

## Model Acceptance Testing

After model data are in PostgreSQL, an instance of the build-time model service is started, and a model service client simulator is launched for model acceptance testing — it replays a month of transactions. Model coverage and accuracy metrics are tracked, and the model build is halted, unless these metrics have acceptable values. On successful test completion, the model is compressed into RPM Package Manager (RPM) package files for distribution (figure 7, step C). The final step is to install the RPMs on a node having hardware matching that used in production, and to again launch the client simulator to replay the transaction history — this time, however, for model latency acceptance testing. Model acceptance testing is split like this for two reasons:

First, latency tests are not reliable unless they are performed using an operating system and hardware-matching production-runtime environment. This is further explained in the next section.

Second, model coverage and accuracy tests do not need production hardware, so these tests are launched right away. If there is a model accuracy or coverage drop (due to, for example, a change in some up-stream system that we do not control), automated tests catch this early.

## Firm Real-Time Deadlines

Some transactions involve counterparties coupled to a small number of other counterparties. These are quick to classify, especially when the counterparties involved are popular. Other transactions involve counterparties weakly coupled to hundreds of counterparties or to counterparties that are relatively rare. Such transactions take longer to classify, because each extra counterparty requires a new b-tree index search, and the more obscure the counterparty, the lower down in the cache hierarchy the coupling table entries for that counterparty are likely to be.

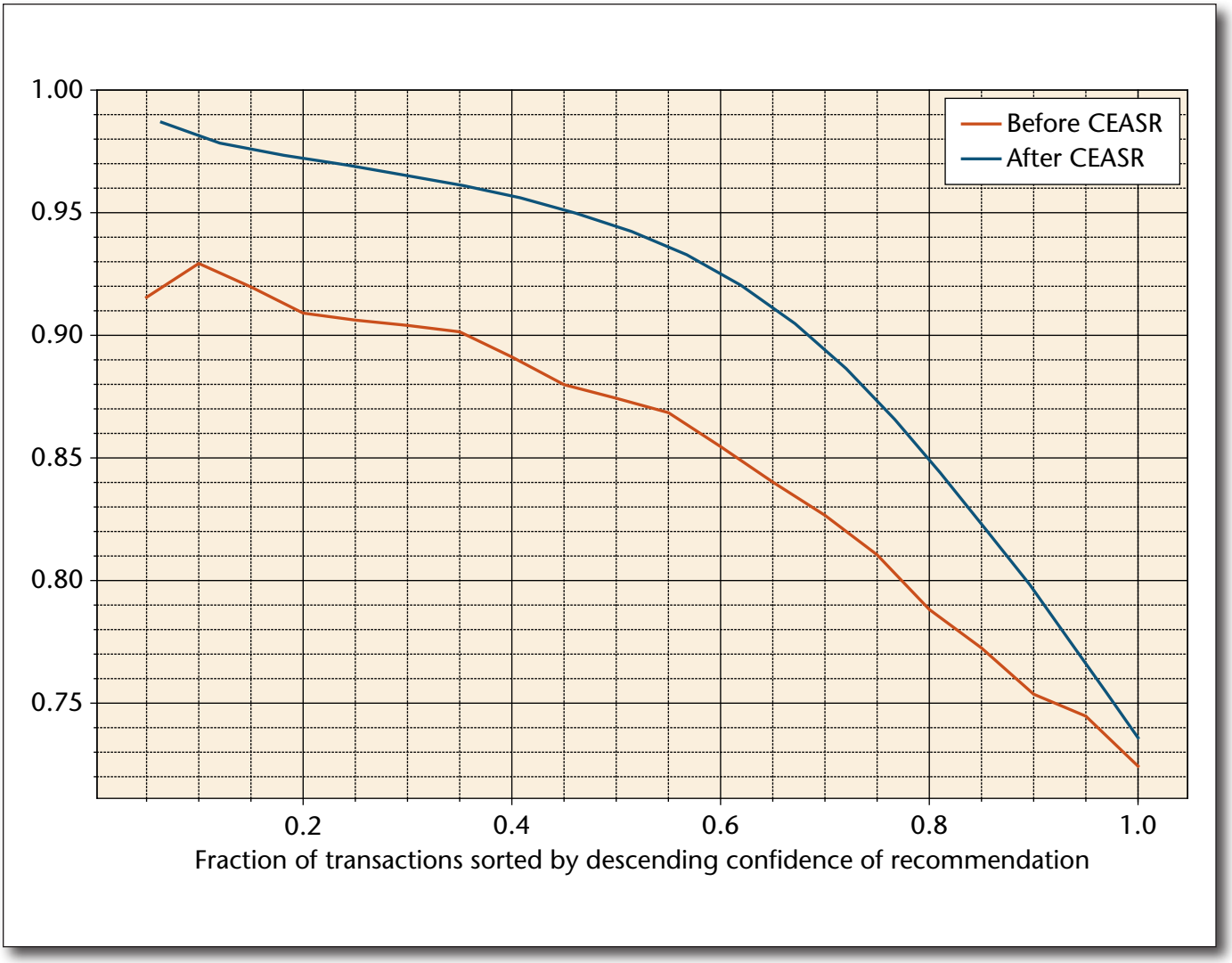


*Figure 4. Accuracy in a Larger User Population Region.*

In our production service, popular counterparties are likely cached, whereas rare counterparties are unlikely to be cached. For this reason, some transactions can take $10^2$ times longer to categorize and models must be latency-tuned to operate predictably under firm real-time deadlines. Deadlines are firm, because failing to show users their transactions on time is worse than if these transactions are missing account recommendations.

## Model Latency Tuning

Latency tuning involves pruning those entries from the model tables that are least likely to influence recommendations. Values that are tiny, for example, are unlikely to make a difference.

With smaller coupling tables, fewer b-tree search steps are needed and a larger portion of the coupling table b-tree index can be cached so index searches are shorter and faster. Yet small coupling tables contain less information, and as the coupling table size is reduced, model coverage and model accuracy both suffer.

During latency tuning, we adjust this trade-off between the model latency (due to coupling-table size) and the model coverage and accuracy. Our goal in latency tuning is to make sure models rarely if ever exceed firm real-time latency deadlines. If a deadline is missed, account predictions are late, so they cannot be used; late predictions — even if correct — are always counted as being incorrect.

Tuning coupling-table sizes for latency also requires that the tuning process sends transaction-request sequences that are representative of what happens in production: transaction counterparties must be as diverse; and counterparty order should be representative.

Latency tuning with just a few transaction counterparties is misleading because, after a counterparty is first referenced, its coupling-table entries are now high up in the cache hierarchy, and subsequent references are quick. A similar cache effect occurs even if you use all possible counterparties but fail to mix up their order. To avoid both of these problems,

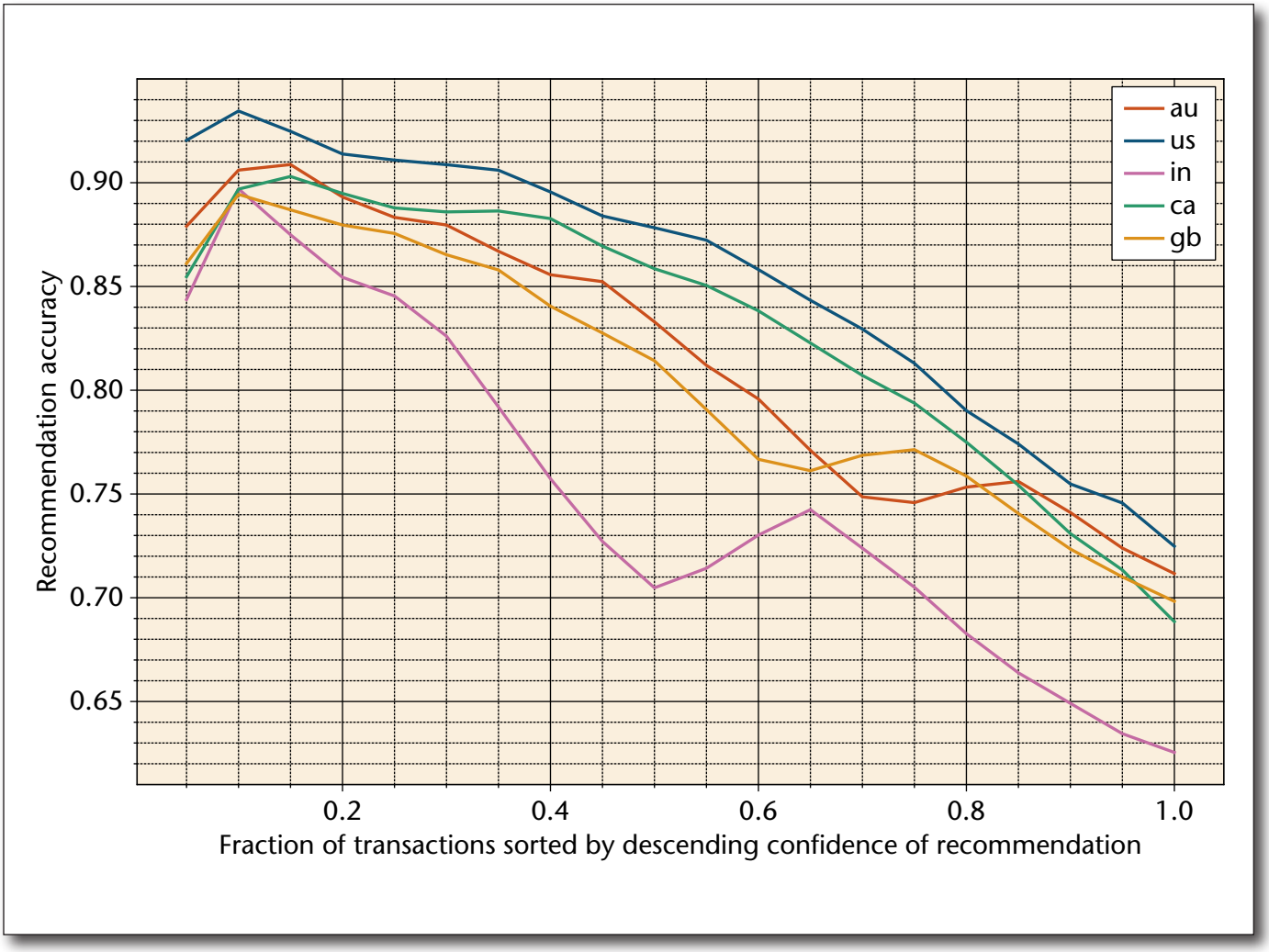


*Figure 5. Accuracy Before CEASR in Various Regions.*

we tune models using sequences of requests that play-back production-model usage from history.

## Build versus Runtime Servers

Our model is regularly refreshed to reflect changes in the real world and comply with regulations such as the European Union's General Data Protection Regulation 2016/679. To enable regular and timely model updates, the build process has to be performance-optimized as well. However, the characteristic patterns of data access during model training are quite different from the interactive context at runtime:

First, model build servers are selected and optimized for sequential large input/output throughput. These have a redundant array of independent disks with small chunks and wide stripes. File systems are created with large records, and operating-system scheduler policies are set to favor throughput over latency. Four-plus central-processing-unit socket servers with nonuniform memory access work well.

Second, model runtime servers are selected and configured to maximize the number of small input/output operations per second. Random-access memory is maximized and solid-state drives are used for model data storage. The file system holding the model is created with small records, and operating-system scheduler policies are set to favor latency over throughput. We avoid nonuniform memory access due to the random access memory-latency overheads it can impose.

Model runtime servers are dedicated for just one task, so no other process competes for input/output or cache. Virtual memory and swap must either be disabled or model process memory must be locked, to prevent being swapped out. This is done so that once a classifier node is running, response latencies stay low and predictable.

## Knowledge Representation

We represent knowledge differently when building models versus when using them. First, during model builds, knowledge is represented inside a column store database (Vertica) using projections in a denormalized format with the same data stored in various sort

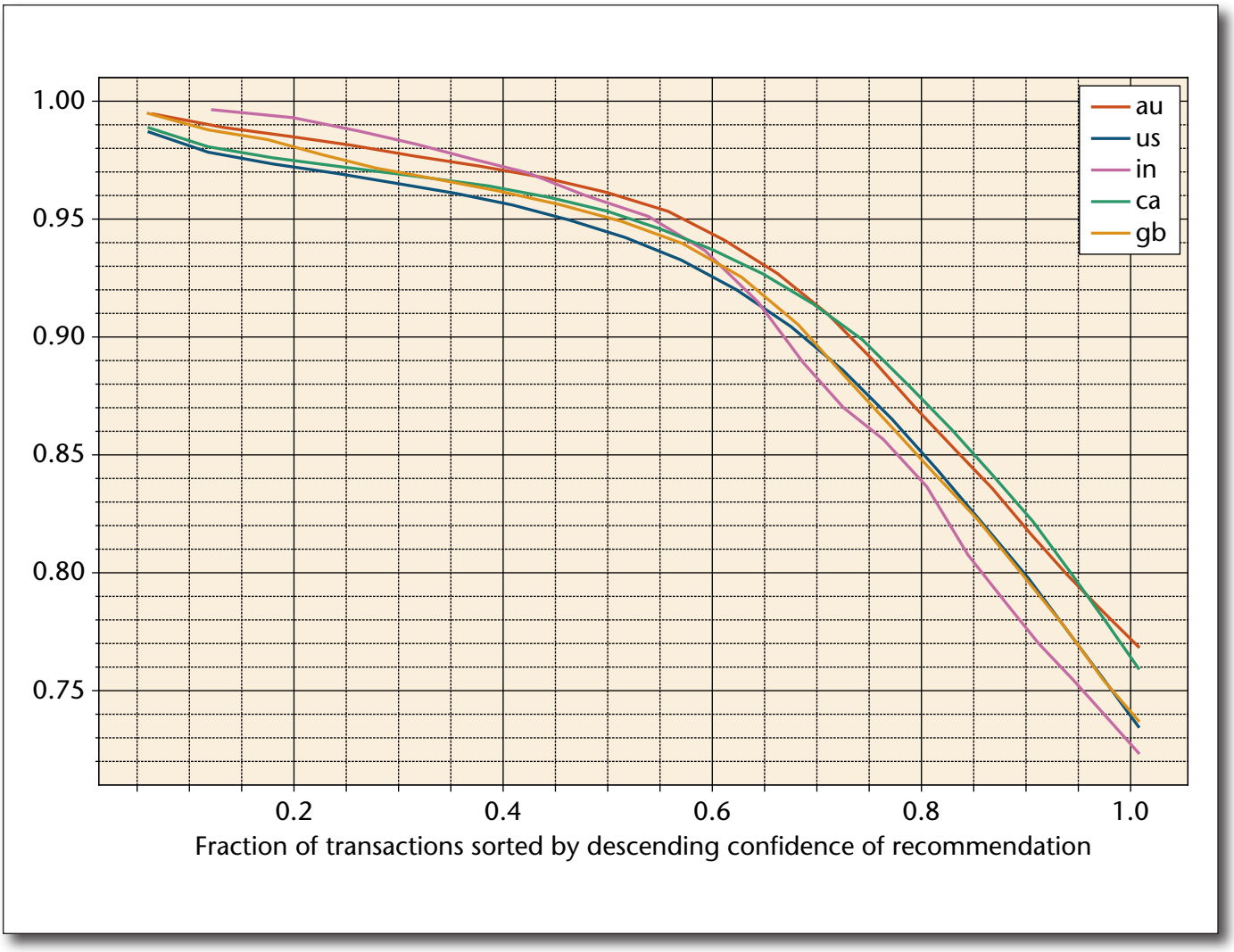


*Figure 6. Accuracy After CEASR in Various Regions.*

orders so that access is sequential, is cache-friendly, and it takes advantage of efficient column-wise compression boosting effective input/output throughput. Second, for model deployment, knowledge is represented using tables in a row-store database (PostgreSQL). Here tables are stored clustered on their primary key, and additional b-tree indexes are built such that the need to access data beyond what is indexed is rare (index-only scans).

The reason for this difference is twofold. First, during model builds, the data-access patterns are known in advance, so that in-memory and on-disk layouts of data can be optimized for cache-hierarchy locality. However, when the model runs in production, we do not know in advance which users, accounts, and counterparties will be involved in any incoming request, hence our knowledge representation must be optimized to answer any request quickly. Second, when the model runs in production, there is a firm real-time latency deadline — requests must be handled in milliseconds because users are waiting; latency concerns dominate over throughput concerns. On the other hand, when a new model is being built, users are not waiting, so latency is not a concern and instead, throughput concerns dominate, because they drive model-refresh cost.

## Fault Causes, Detection, and Recovery

Data extraction from our data warehouse is vulnerable to unexpected changes in how the accounting products we support represent their information. Database schema changes that cleanly break data-extraction scripts are straightforward to detect. Harder to detect are shifts in the meaning of the same database schema, such as happens when upstream product teams make an effort to minimize database schema changes. In these cases, data-extraction scripts may yield incomplete sets of training transactions, and other unexpected and undesirable results.

Another source of faults is Apache Hive,[4] which is used inside our data warehouse. Hive is not self-tuning — thus, as the amount of data grows and the

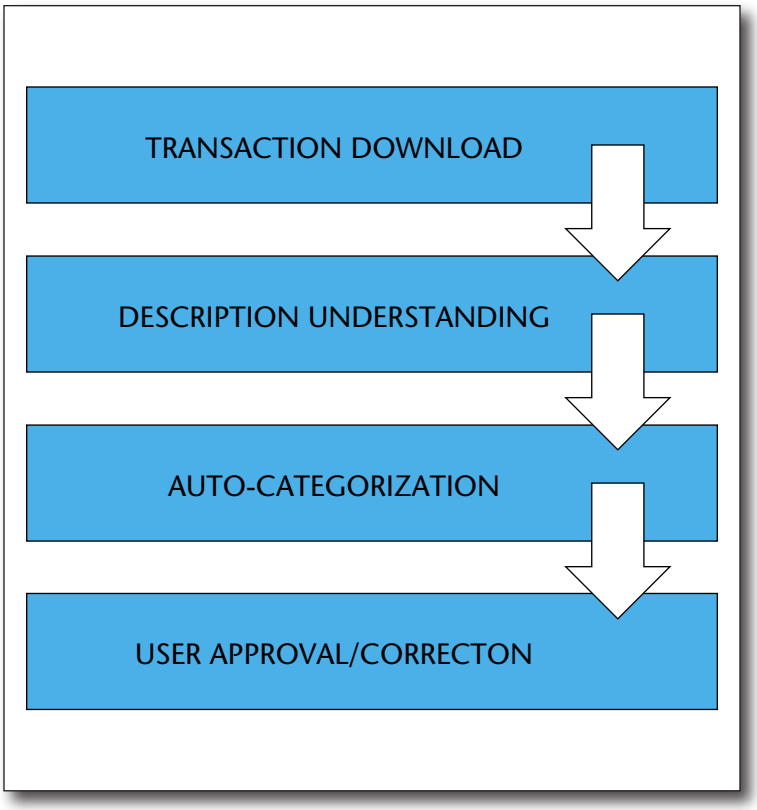


*Figure 7. Stages of Financial Transaction Processing.*

distribution of that data changes, Hive queries that once ran fine can take an exorbitant amount of time, or outright fail. For example, query plans that poorly spread out processing across the cluster can cause too much data to be sent to just a few nodes. With their memory exhausted, nodes crash, and manual intervention is required to restructure the query to prevent this happening in the future.

Tables with billions of rows are created, exported, imported, and indexed during each model build so that operations such as *table creation* or *indexing* are split across multiple workers; and when exporting or importing tables, we divide them into chunks that are handled in parallel — without chunking, large tables limit scalability.

As a consequence, a model build involves many pools of workers — easily adding up to hundreds of workers. While the chance of any one worker encountering problems is small because the work is carried out by so many, it is essential to consider what happens when faults occur. Specifically, when manual intervention is required, how do you bring hundreds of parallel workers to a state from which fault diagnosis and fault recovery is possible?

Our technique is to have each task worker carry out the following steps, listed in order:

Step 1: Check that no other worker has reported a fault. If a build-halting fault has been reported, no worker will start new work. The worker that encounters a build-halting fault does not forcibly kill other concurrent workers, because this would leave many partially complete tasks needing cleanup; thus, we let any already-running tasks finish gracefully.

Step 2: Check that the assigned task has not already been performed. If a checkpoint has already been committed for this task, indicating it has already been performed and its results have been verified, the worker will log this fact, but will otherwise do nothing, and its worker pool will assign to it a different task.

Step 3: Check that conditions necessary to start the assigned task are satisfied. A worker that copies table chunks from our column-store database to our row-store database, for example, checks that source and target databases are alive, that source and target schema exist, and that source and target tables exist. If any of these conditions are not satisfied, the worker reports the fault and exits.

Step 4: According to the previous three steps, if no other worker has reported a fault, and the assigned task has not been performed, then the conditions necessary to start the task are satisfied. If all of these are true, we also check whether partial results from a previous task attempt exist and, if needed, a cleanup is performed. Finally, the assigned task is started.

Step 5: After the task is done, assertion checks are used to verify that the task finished correctly. For example, if the task is to launch a database, we check that the database answers client connections, and allows expected operations. Another example: if the task is to build a certain table, when that task is done, we verify that the table exists and that it contains a reasonable number of rows and sanity-check other such attributes about it. A task is considered complete and its checkpoint is committed if, and only if, all assertions about it pass — otherwise the worker reports a fault, and exits.

This step-wise technique prevents fault cascades. A badly created table does not, for example, cause other tables to be badly created, because we catch it right away. The technique also allows fault recovery. Once a fault is cleared, if ninety percent of a multi-day model build is complete, when the build is restarted, just the remaining ten percent of tasks will be attempted due to checkpoints from a previous run.

## Model Deployment

Our model operates as a service application–programming interface deployed using a cluster of identical classifier nodes behind a load balancer. Incoming requests first go to the load balancer, which then forwards the request to an available classifier node. If the continuous load on the least-busy classifier node is too high, additional classifier nodes are added. If the continuous load on the busiest classifier node falls, the oldest classifier node is removed from the load balancer pool and stopped. If a classifier node malfunctions (for example, timeouts on requests) the load balancer automatically replaces it with a new node, thus healing the service. This healing

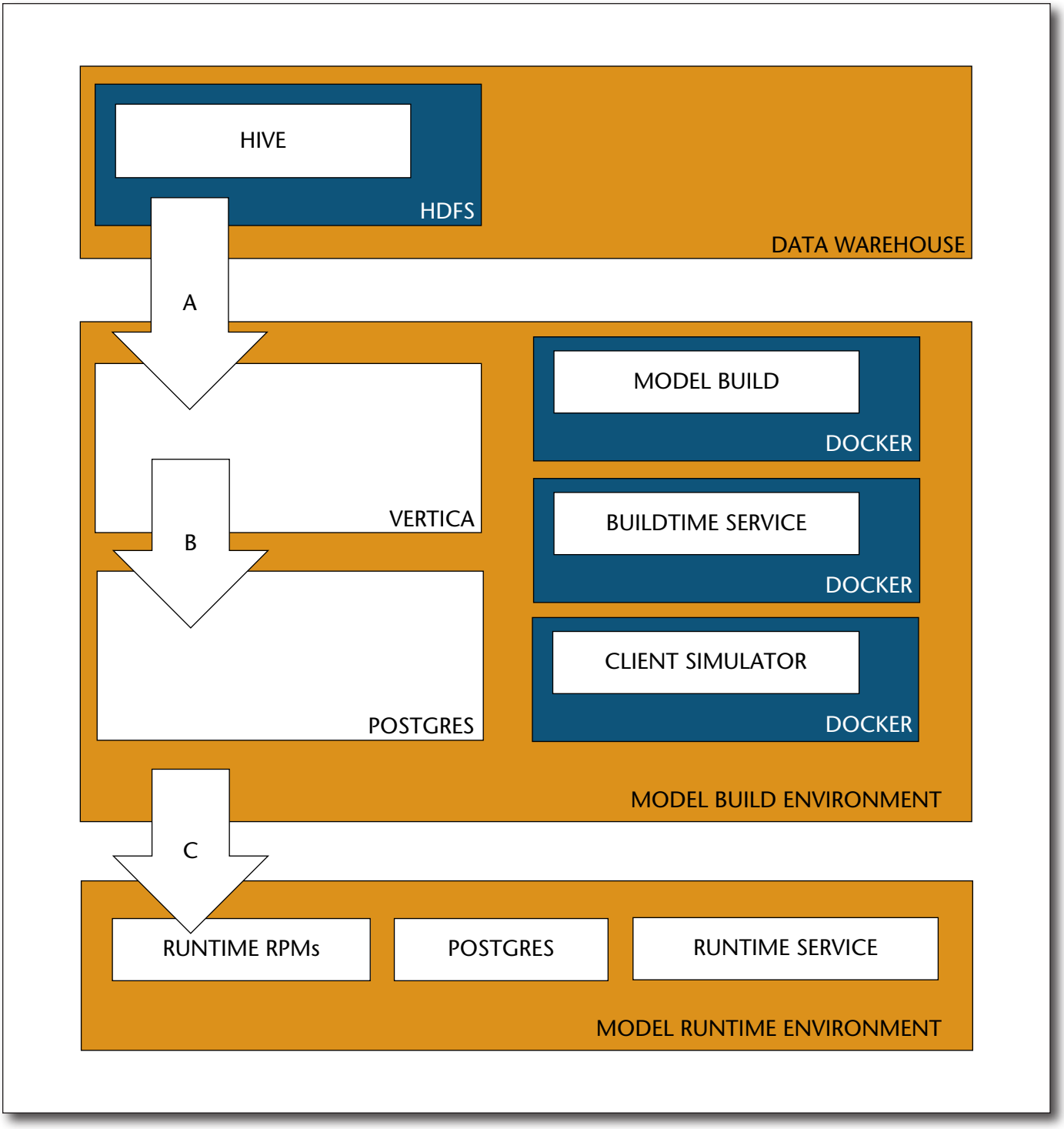


*Figure 8. Model Build Environment.*

functionality is also used for zero downtime upgrades such as when fresh models are deployed — old classifier nodes are purposefully terminated one at a time, and the load balancer replaces them with upgraded versions.

We use a shared-nothing architecture because it makes service testing, deployment, and scaling straightforward. For example, when the number of incoming requests doubles, the number of running classifier instances is approximately doubled. When the number of incoming requests drops in half, the number of running classifier nodes is approximately dropped in half. The ratio is approximate because classifier startup takes several minutes, so extra classifier nodes are always kept around to handle spikes in demand.

## User Impact and Benefits

Improvements in the ability to accurately categorize financial transactions are of significant economic value. For a sense of scale, if — without automation — it takes three seconds to select the proper CoA account for a financial transaction, last year the users of our accounting software would have spent well over 1,000 man years on this task.

The first version of our ML-based financial transaction categorization service was deployed to production for English-language QuickBooks (USA, Canada, UK — more than two million users) in November 2016. Non-English user regions (such as France and India) were added in August 2017. Compared with the legacy systems that it replaced, the service was able to classify more transactions and to do so at a higher accuracy — it had fifty-six percent fewer uncategorized transactions and twenty-eight percent fewer errors, representing a remarkable reduction in the amount of manual work that millions of QuickBooks users have to do to file their financial transactions.

In 2019, we developed a framework for personal categorization called CEASR. Comparing 2019 performance against our 2018 performance, we see significant improvement in the overall accuracy across multiple user regions. We also see an improvement in our ability to sort recommendations by expected accuracy, which lets our users focus their attention on an even smaller fraction of low-confidence transactions while automation handles everything else.

## Conclusions

Building on the work by Lesner et al. (2019), we have presented our improved approach for personalized classification of financial transactions to automate accounting. The improved approach merges the two common supervised machine-learning paradigms of classification and recommendation systems into a single framework that can flexibly incorporate propositional and relational representation of the domain, and is efficient for dealing with high-cardinality nominal attributes, variable and changing numbers of classes, and evolving class definitions.

We shared lessons learned with respect to differences in data-access patterns during model training and runtime-production deployment. We explained how these differences can be effectively supported by adopting a column-oriented data format for model training and a row-oriented format for runtime deployment. We also discussed the requirements related to real-time constraints on model runtime performance, and suggested ways to satisfy such constraints.

Our system has been deployed at scale, and handles billions of financial transactions for millions of small businesses each year. Fragments of information from millions of users are combined in a manner that allows us to accurately recommend user-specific CoA accounts. CoA accounts are handled even if named using abbreviations or in a foreign language. Transactions are handled even if a given user has never categorized a transaction like that before. The development of such a system and testing it at scale over billions of transactions is a first in the financial industry.

## Notes

1. https://quickbooks.intuit.com/
2. https://www.vertica.com/
3. https://www.postgresql.org/
4. See hive.apache.org

## References


Clark, K., and Manning, C. D. 2015. Entity-Centric Coreference Resolution with Model Stacking. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL 2015)*, Volume 1: Long Papers, 1405–15. Stroudsberg, PA: Association for Computational Linguistics.

Efron, B., and Tibshirani, R. J. 1998. *An Introduction to the Bootstrap*, Volume 57: Applied Probability. London, UK: Chapman & Hall/CRC.

Freund, Y., and Schapire, R. E. 1995. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting*, 23–37. Berlin, Germany: Springer.

Grove, A. J., and Schuurmans, D. 1998. Boosting in the Limit: Maximizing the Margin of Learned Ensembles. In *Proceedings of the Fifteenth Association for the Advancement of Artificial Intelligence (AAAI) Conference, 692–9*. Menlo Park, CA: Association for the Advancement of Artificial Intelligence (AAAI) Press/The Massachusetts Institute of Technology (MIT) Press.

Lesner, C.; Ran, A.; Rukonic, M.; and Wang, W. 2019. Large scale Personalized Categorization of Financial Transactions. In *Proceedings of the Thirty-Third Advancement of Artificial Intelligence (AAAI) Conference, 9365–72*. Palo Alto, CA: Association for the Advancement of Artificial Intelligence (AAAI) Press. doi.org/10.1609/aaai.v33i01.33019365.

Li, L.; Hu, Q.; Wu, X.; and Yu, D. 2014. Exploration of Classification Confidence in Ensemble Learning. *Pattern Recognition* 47(9): 3120–31. doi.org/10.1016/j.patcog.2014.03.021.

Melnik, O.; Vardi, Y.; and Zhang, C.-H. 2004. Mixed Group Ranks: Preference and Confidence in Classifier Combination. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26(8): 973–81. doi.org/10.1109/TPAMI.2004.48.

Micci-Barreca, D. 2001. A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems. *SIGKDD Explorations* 3(1): 27–32. doi.org/10.1145/507533.507538.

Montalbo, F. J. P., and Festijo, E. D. 2019. Comparative Analysis of Ensemble Learning Methods in Classifying Network Intrusions. In *2019 Institute of Electrical and Electronics Engineers* (IEEE) *Conference on System Engineering and Technology, 431–6*. Piscataway, NJ: Institute of Electrical and Electronics Engineers (IEEE). doi.org/10.1109/ICSEngT.2019.8906310.

Recchia, G., and Nulty, P. 2017. Improving a Fundamental Measure of Lexical Association. In *Proceedings of the 41st Annual Meeting of the Cognitive Science Society*. Seattle, WA: The Cognitive Science Society.

Vechtomova, O. 2009. Review of *Introduction to Information Retrieval* by Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Computational Linguistics* 35(2): 307–9. doi.org/10.1162/coli.2009.35.2.307.


Wang, Y.; Li, D.; Liu, H.; and Choi, I.-C. 2015. *Generalized Ensemble Model for Document Ranking*. abs/1507.08586. Ithaca, NY: Cornell University Library.

**Christopher Lesner** is a principal software engineer at Intuit, where he builds Intuit's machine-learning systems that analyze billions of financial transactions each year for small business accounting and small business loans. Before Intuit, Lesner developed fault-tolerant, real-time distributed software for telecom billing, interactive voice response, short message service voting, and network configuration management. Christopher has two decades of software design, development, and deployment experience and holds an MS in computer science from McMaster University. He is an inventor on 50 patents and coauthor of several peer-reviewed technical publications.

**Alexander Ran** is a distinguished software engineer at Intuit, where he leads a team of data scientists and engineers that developed Intuit's industry-leading financial transactions classification system used for small business accounting and small business loans. Before Intuit, Ran was a Research Fellow at the Nokia Research Center, where he led research in natural language processing and software architecture. He is an inventor on more than 60 patents and coauthor on more than 30 peer-reviewed technical publications. Ran has an MS in Physics from the Hebrew University of Jerusalem and a PhD in Computer Science from the Helsinki University of Technology.

**Wei Wang** is a data scientist at Intuit, where he applies statistical and machine-learning algorithms to understand business events revealed by financial transactions for small business accounting and credit scoring. Wang is a member of the team that developed and tested the algorithms used by QuickBooks Capital last year for tens of millions of dollars in lending decisions. Before Intuit, Wang was a statistician at Vanderbilt University and the Palo Alto Medical Foundation Research Institute, where he designed experiments and analyzed data from medical imaging and clinical trials to build predictive models for disease prognosis. He has an MS in Applied Statistics and a PhD from Michigan State University.

**Marko Rukonic** is a principal software engineer at Intuit, where he builds Intuit's machine-learning systems that analyze billions of financial transactions each year. Rukonic has two decades of expertise in the design, development, and deployment of high-reliability distributed software for online processing of financial data and payments. Marko holds a Master of Science degree in Electrical Engineering and Computer Science from the University of Zagreb, and is an inventor on more than 20 patents and a coauthor of several peer-reviewed technical publications.