WHITE PAPER

Redis for Machine Learning

Tague Griffith, Redis

CONTENTS

Executive Summary

Machine learning is fast becoming a critical requirement for modern smart applications. Predictive analytics are being implemented across all industries in use cases ranging from fraud detection and product recommendations to demand forecasting and sentiment analysis.

Companies that ignore advances in machine learning risk irrelevance, but effectively implementing the technology requires considerable resources and effort. As a result, many companies are forced to make painful trade offs between delivering the high performance their application users have come to rely upon and delivering predictions that improve the overall product.

In this white paper, we'll overview the challenges developers face when implementing machine learning solutions and present Redis-ML, an add-on Redis module, as an antidote to these challenges. A deep dive into the specific features of Redis-ML, along with how these features leverage the high performance and flexibility you've come to expect with Redis, will make a compelling case for building real-time predictive engines in Redis-ML.

Challenges Deploying Machine Learning Solutions

The majority of machine learning toolkits focus primarily on the act of training models. When it comes to deploying these models within a production environment, there are far fewer out-of-the-box resources available, leaving developers to deal with the significant challenges of deploying custom machine learning solutions. These challenges include:

- Real-time requirement
- High availability assurance
- Live data testing
- A lack of language interoperability
- The esoteric nature of machine learning frameworks

Real-Time Requirement

Training new machine learning models can be done offline. However, in order for your applications to maximize the benefits of machine learning, predictive intelligence should be delivered in real time. Many of today's most exciting machine learning use cases (e.g. fraud detection, product recommendations, self-driving cars) are only effective if correlations are discovered and acted upon on the fly.

This real-time requirement means that applications must be capable of serving very complex models at scale, without missing a beat. Unfortunately, given the machine learning platforms currently on the market, this is often an unattainable goal. Developers are either forced to make performance trade offs or implement machine learning functionalities as offline or batch exercises.

High Availability Assurance

Fault tolerant, highly available systems are difficult to build. But without out-of-the-box machine learning platforms that natively support automated failure detection and failover, companies must dedicate significant resources and expertise to ensure their applications reliably deliver real-time predictive intelligence.

Live Data Testing

Newly trained models need to be evaluated against live data to ensure the validity of their operation. Updating models on the fly is difficult to implement, often, the process for doing so doesn't exist or can only happen in the test environment.

Lack of Language Interoperability

Another common challenge is lack of language interoperability. Often, machine learning models can only be retrieved, up-

dated, or executed by applications that are written in the same language as they are because their databases aren't capable of storing these models in their native formats.

Esoteric Nature of Machine Learning Frameworks

Chances are, any custom or out-of-the-box machine learning framework deployed will be highly esoteric and therefore unfamiliar to your operations staff. This lack of at-the-ready expertise will require additional resources (time, training, skills) and hinder the ability to overcome other deployment challenges.

Introducing Redis-ML

Redis-ML is a Redis module that implements several machine learning models as built-in Redis data types. Unlike almost all other machine learning frameworks, Redis-ML supports turnkey use of those models for real-time predictions.

Using Redis-ML, machine learning trained models can be loaded from any platform and are immediately ready to serve in a production environment. This fills the gap between the learning and predictive sides of machine learning, and allows developers to build their applications within a familiar, full-featured data store rather than having to build homegrown services.

Open Source Software

Redis-ML has a general open source license, allowing it to be easily extended to meet unique customer needs. Available for download from GitHub (see https://github.com/RedisModules/redis-ml), the module—and its users—benefit from the support and innovation of one of the largest and most engaged developer communities.

Compatible with Popular Machine Learning Libraries

Redis-ML is toolkit agnostic. It stores and serves models from many different machine learning libraries such as Apache Spark and scikit-learn.

Due to the popularity of the Apache Spark machine learning library, known as Spark ML, Redis offers a specialized module for connecting Spark and Redis. Available for download from GitHub (see https://github.com/Redis/spark-redis-ml), the Spark-Redis-ML package is a Redis connector for Apache Spark that provides read and write access to all of Redis' core data structures as RDDs (resilient distributed datasets).

Redis-ML Features & Tutorials

In this section, each of the primary features supported by Redis-ML will be discussed in detail, complete with sample scenarios and code that you are invited to follow along with. These features include:

- Linear regression
- Logistic regression
- Matrix operations
- Decision trees

Setting Up the Tutorial Environment

The sample code used throughout this section's tutorials is written in Python 3 using a variety of freely available packages for machine learning. You will need to install the following packages, using pip3 or your preferred package manager, to run the samples:

- scikit-learn (0.18.2)
- NumPy (1.13.1)

- SciPy (0.19.1)
- Redis (2.10.5)

You will also need a Redis 4.0.0 instance and the Redis-ML module. Shay Nativ, the developer behind the Redis-ML module, created a Docker container with Redis 4.0.0 and the Redis-ML module preloaded. To use this container in conjunction with the sample code, launch the container using the command:

```
docker run -it -p 6379:6379 shaynativ/redis-ml
```

Docker will automatically download and run the container, mapping the default Redis port (6379) from the container to your computer. The -p option maps the port 6379 on the host machine to port 6379 in the container, so take the appropriate security precautions for your environment.

Linear Regression

Linear regression has been part of the statistician's toolbox long before algorithmic machine learning was invented. This modeling technique attempts to predict a result (often referred to as the dependent value) from one or more known quantities (explanatory variables). For linear regression to work, we must be able to accurately estimate our results with a straight line.

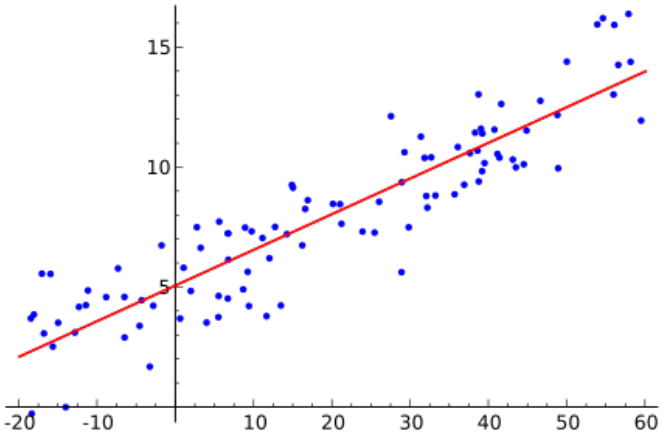

Figure 1. Example of simple linear regression with one independent variable (Credit: Linear regression. In Wikipedia, The Free Encyclopedia.)

In the previous graph, notice how our data points cluster around an idealized line. This is what makes this sample dataset a good candidate for linear regression. In practice, linear regression is used to model a variety of real-world scenarios in which linear relationships from observations can accurately predict outcomes, such as the price of a house based on square footage, or college GPA based on high school GPA and SAT scores.

Tutorial 1: Predicting Median House Prices Based on Average Number of Rooms

In this tutorial, we will walk through the code for a sample program for predicting median housing prices from various features of a neighborhood.

> *Note: The sample code in this tutorial is written in Python and requires a Redis instance running Redis-ML. The instructions for setting up the appropriate Redis environment are identical for all tutorials in this white paper and can be found in the overview of the Redis-ML Features section.*

Part I: Learning the Model

From algebra, we know that a line is represented by the standard equation $y=b+ax$, so to "learn" a model of this form, we need to apply an algorithm to discover the parameters of the line—the slope and intercept. Nothing incredibly fancy; in fact, prior to algorithmic machine learning, most statisticians would "fit" these models by hand. These days, it's far more common to use a computer to find the line's parameters, and a variety of toolkits (e.g. TensorFlow, scikit, Apache Spark) are available to solve linear regression problems.

The important thing to remember is that once we've learned a linear regression model, we have an agnostic mathematical formula for predicting results that can be implemented by any system.

Let's work through an example of performing a linear regression and discovering the model parameters using the popular Python scikit-learn package and the Boston Housing dataset, which is commonly used to teach statistics and machine learning. This dataset predicts the median housing price for neighborhoods in the Boston area using neighborhood features such as the average number of rooms in a house, the distance from main Boston employment centers, or the crime rate. To make it easier to visualize the linear regression process, we're going to work with a single data feature: the average rooms per dwelling (RM) column.

Let's start by plotting our data to visualize the relationship between room count (RM) and median price (MEDV):

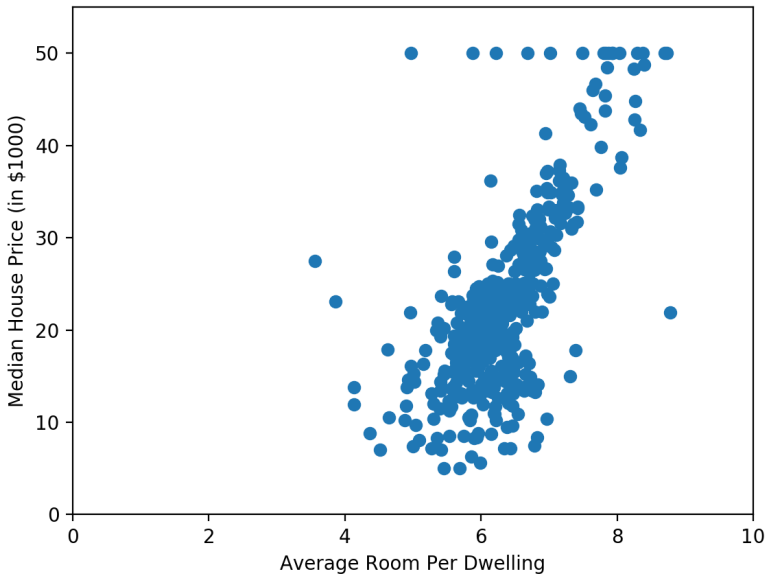

Figure 2. Relationship between median house price and average number of rooms.

While not a perfect line, we can see a pretty strong linear relationship between the average number of rooms and the median house price. As shown next, we can even draw an idealized representation of the relationship and see how the data points cluster around it.

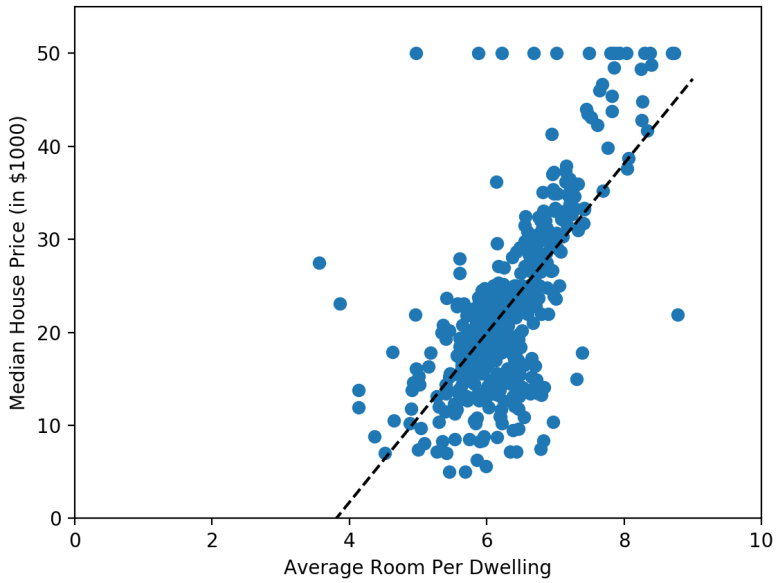

Figure 3. Idealized linear relationship between median house price and average number of rooms.

The following code demonstrates how to load the Boston Housing dataset using scikit. The Boston Housing dataset consists of twelve different features used to predict housing prices, but we want to use only one for simplicity. So, after loading the dataset, we want to extract data from the fifth column (the RM column which represents average number of rooms).

```
from sklearn.datasets import load_boston

boston = load_boston()

boston_RM = boston.data[:,5]
boston_PRICE = boston.target
```

Now we split our data into two sets: a training set and a test set. Let's create our training set from the first 400 samples and our test set from the remaining 106 samples. This method of splitting ensure ensures we always run with the same sets for reproducible results.

```
# slice the data into train and test sets
x_train = boston_RM[:400].reshape(-1, 1)
x_test =  boston_RM[400:].reshape(-1, 1)
y_train = boston.target[:400]
y_test = boston.target[400:]
```

Now that we've constructed our training and test sets, we can use the LinearRegression model supplied by scikit to fit a line to our data.

```
lm = LinearRegression()
lm.fit(x_train, y_train)

coef = lm.coef_[0]
int = lm.intercept_
```

```
print('Coef: {coef}, Intercept: {int}'.format(coef=coef, int=int))
```

After running our code, we find that scikit has fit a line to our data with a coefficient of 9.40550212 and an intercept of -35.26094818316348.

> ***Note about toolkit independence:*** *In this tutorial, we are using Python to run a linear regression over sample data using the scikit-learn package. But it's important to note that any toolkit capable of performing a linear regression on a dataset to discover the parameters of the line can be used in conjunction with Redis. In other words, it is the model—not the toolkit—that is important when using Redis to serve your models. From this point on, the procedure for setting up Redis to predict housing prices is nearly identical regardless of the toolkit used to discover the parameters of the line.*

Part II: Building a Real-time Prediction Engine Using Redis 4.0.0 and the Redis-ML Module

Now that we have the parameters of our line, we can implement a linear regression model to predict housing prices in the Boston area based on the average number of rooms in a house in a neighborhood of interest. Great! But an important question remains: how can you build an application to make real-time predictions and use the functionality as an app or a website?

The scikit package provides a predict function to evaluate a trained model, but using a function within an application requires the implementation of a host of other services in order to make it fast and reliable. This is where Redis can augment your machine learning systems.

The Redis-ML module takes advantage of the new Modules API to add a standard linear regression as a native datatype. The module can create linear regressions as well as use them to predict values.

To add a linear regression to Redis, you need to use the `ML.LINREG.SET` command to add a linear regression to the database. The `ML.LINREG.SET` command takes the following form:

```
ML.LINREG.SET key intercept coeef [...]
```

By convention, all of the commands in the Redis-ML module begin with the module's identifier, ML. All linear regression commands are prefixed with LINREG.

To set up Redis to be a predictive engine for Boston housing prices using the line we fit in scikit, we need to first load the Redis-ML module using the loadmodule directive.

```
redis-server --loadmodule /path/to/redis-ml/module.so
```

Then we set a key to represent our linear regression using the constants from scikit by executing the `ML.LINREG.SET` command. Remember that the intercept is the first value supplied and the coefficients are provided in feature order. From our scikit code, we determined our line had a coefficient of 9.40550212 for the *RM* variable and an intercept of -35.26094818316348. We can use the `ML.LINREG.SET` command to set a Redis key to compute this linear relationship.

```
127.0.0.1:6379> ML.LINREG.SET boston_house_price:rm-only
-35.26094818316348 9.40550212
OK
```

Once our `boston_house_price:rm-only` key is created, we can repeatedly predict the median house price in a neighborhood by using the `ML.LINREG.PREDICT` command. For example, to predict the median house price in a neighborhood that averages 6.2 rooms per house we would run the command:

```
127.0.0.1:6379>  ML.LINREG.PREDICT boston_house_price:rm-only 6.2
"23.053164960836519"
```

Redis predicts a median house price of $23,053 (remember our housing prices are in thousands) for this neighborhood.

While it's helpful to understand how to work with the `ML.LINREG` commands from the redis-cli, it is far more likely we would be doing this from an application. We can extend our Python code which fits the regression line to automatically create the `boston_house_price:rm-only` key in Redis. Once we've created the key in Redis, we implement a test that generates predictions from Redis using our test data.

```
r = redis.StrictRedis('localhost', 6379)
r.execute_command("ML.LINREG.SET", "boston_house_price:rm-only",
"-35.26094818316348", "9.40550212", )

redis_predict = []
for x in x_test:
      y = r.execute_command("ML.LINREG.PREDICT", "boston_house_price:rm-only", x[0])
      redis_predict.append(float(y))
```

We can also generate scikit's predictions for the same set of data using the predict routine:

```
y_predict = lm.predict(x_test)
```

For comparison, we've plotted the results. In the following graph, the black circles represent the actual prices for the test data in our dataset. The blue markers (+) represent the values predicted by scikit and the magenta markers (x) represent the values predicted by Redis.

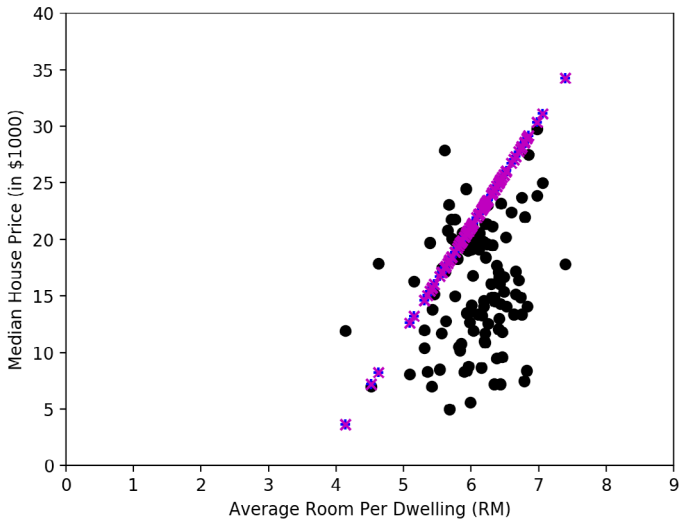

Figure 4. Redis and sc) represent the values predicted by scikit and the magenta markers (x) represent the values predicted by Redis).

As you can see, Redis and scikit make the same predictions for median house price given the average number of rooms. While linear regression may not correctly predict the exact price of every data point, it provides a useful means of estimating an unknown price based on observable features of a home or neighborhood.

Tutorial 2: Predicting Median House Prices Based on Multiple Variables

In the previous linear regression tutorial, we used a single variable to predict a value. But linear regression is often performed with multiple variables as the predictors for a single value, called multiple linear regression. This gives us a model that looks like $y=\beta_0 + \beta_1 x_1 + ... + \beta_n x_n$. In a multiple linear regression, we need to solve for an intercept and coefficients for each variable.

Note: To ensure a thorough understanding of the concepts and commands highlighted in this tutorial, it is recommended that you first perform the previous linear regresstion tutorial found in this section.

The following code implements a multiple linear regression, fitting a prediction line using all of the available data columns from our Boston housing dataset:

```
import redis

from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# load out data
boston = load_boston()

# slice the data into train and test sets
x_train = boston.data[:400]
x_test =  boston.data[400:]
y_train = boston.target[:400,]
y_test = boston.target[400:,]

# fit the regression line
lm = LinearRegression()
lm.fit(x_train, y_train)
y_predict = lm.predict(x_test)

coef = lm.coef_
inter = lm.intercept_

for col, c in zip(boston.feature_names, coef):
      print('{colname:7}:\t{coef:0.6f}'.format(colname=col,coef=c))
print("Intercept: {inter}".format(inter=inter))

# set the linear regression in Redis
cmd = ["ML.LINREG.SET", "boston_house_price:full"]
cmd.append(str(inter))
cmd.extend([str(c) for c in coef])

r = redis.StrictRedis('localhost', 6379)
r.execute_command(*cmd)

cmd.append(str(inter))
cmd.extend([str(c) for c in coef])
r = redis.StrictRedis('localhost', 6379)
r.execute_command(*cmd)
```

The table below shows the coefficient that scikit determined for the best predictor line. Each coefficient corresponds to a particular variable (feature). For example, in our multiple linear regression, the constant for average rooms is now 4.887730, instead of the 9.4055 that resulted when only considering the room data.

```
CRIM    :       -0.191246

ZN      :       0.044229

INDUS   :       0.055221

CHAS    :       1.716314

NOX     :       -14.995722

RM      :       4.887730

AGE     :       0.002609

DIS     :       -1.294808

RAD     :       0.484787

TAX     :       -0.015401

PTRATIO:        -0.808795

B       :       -0.001292

LSTAT   :       -0.517954

Intercept: 28.672599590856002
```

From here, we create a Redis key `boston_house_price:full` to store our represented multiple linear regression. Keep in mind that Redis doesn't use named parameters for the arguments to the `ML.LINREG` commands. The order of the coefficients in the `ML.LINREG.SET` must match the order of the variable values in the `ML.LINREG.PREDICT` call. As an example, using the linear regression code above, we would need to use the following order of parameters to our `ML.LINREG.PREDICT` call:

```
127.0.0.1:6379>  ML.LINREG.PREDICT boston_house_price:all <CRIM> <ZN> <INDUS> <CHAS>
<NOX> <RM> <AGE> <DIS> <RAD> <TAX> <PTRATIO> <B> <LSTAT>
```

Logistic Regression

Logistic regression is another type of linear model for building predictive models from observed data. Unlike linear regression, which is used to predict a value, logistic regression is used to predict binary values (e.g. pass/fail, win/lose, healthy/sick). This makes logistic regression a form of classification. The basic logistic regression can be augmented to solve multiclass classification problems.

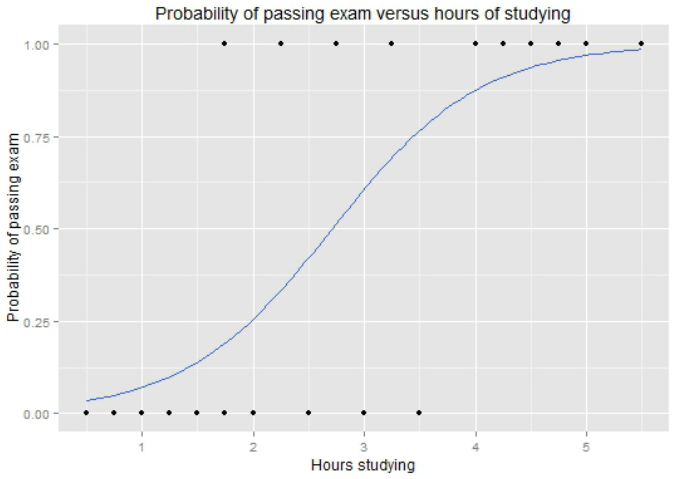

Figure 5. Graph of a logistic regression curve showing the probability of passing an exam relative to hours spent studying (Credit: Logistic regression. In Wikipedia, The Free Encyclopedia.)

The above graph shows a plot of the probability of passing an exam relative to the hours spent studying. Logistic regression is a good technique for solving this problem because we are attempting to determine pass/fail, which is a binary selector. If we wanted to determine a likely grade or percentage on the exam, simple linear regression would be a better technique.

Tutorial: Classifying Iris Plants into Three Different Species

To demonstrate logistic regression and how it can be implemented in Redis, we will walk through the code for a sample program that classifies Iris plants into three different Iris species.

> *Note: The sample code in this tutorial is written in Python and requires a Redis instance running Redis-ML. The instructions for setting up the appropriate Redis environment are identical for all tutorials in this white paper and can be found in the overview of the Redis-ML Features section.*

Part I: Learning the Model

Let's work through an example of performing a logistic regression and discovering the model parameters using another classic data set available in the Python scikit-learn package: Fisher's Iris dataset.

The Fisher's Iris dataset consists of 150 data points, each labeled as one of three different species of Iris: Iris setosa, Iris versicolor, and Iris virginica. Each data point is made up of four attributes (i.e. features) of the plant. Using logistic regression, we can use these attributes to classify an Iris into one of the three species.

Load the Fisher's Iris dataset using scikit.

```
from sklearn.datasets import load_iris
iris = load_iris()
```

If we view the data in table form, we can see that the four attributes are sepal length and width (in case you were curious, the sepal is the outer part of the flower that encloses a developing bud and supports the blooming petals), and petal length and width.

Sepal length	Sepal width	Petal length	Petal width
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.8	3.1	1.5	0.2
5.0	3.6	1.4	0.3
5.4	3.9	1.7	0.4

Figure 6. First six rows of Fisher's Iris dataset (Credit: Iris flower data set. In Wikipedia, The Free Encyclopedia.)

Our target classification is encoded as integer values 0, 1, and 2 with the following mapping:

0	Iris setosa
1	Iris versicolor
2	Iris virginica

To get a better sense of the relationship between various sepal and petal measurements and corresponding flower type, we generated two plots: one of sepal width versus length and another of petal width versus length.

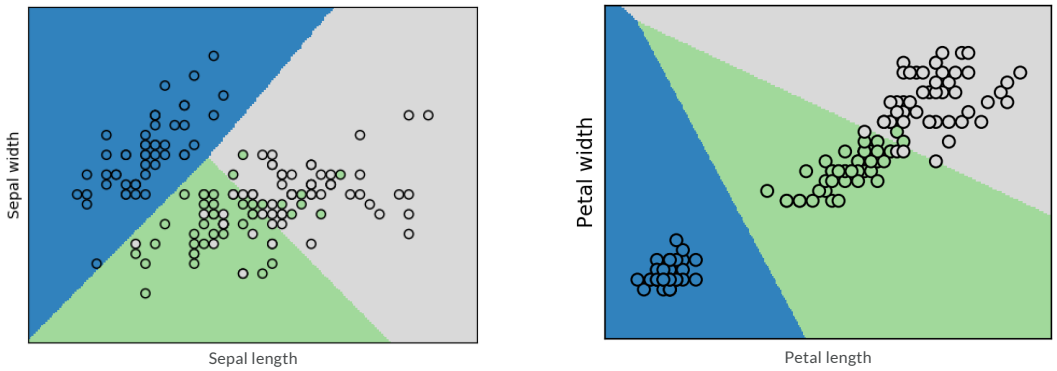

Figure 6. Plots classifying Iris types according to sepal measurements (left) and petal measurements (right).

Both of the previous graphs maps the classification boundaries (determined through logistic regression) of the three classes and overlay these boundaries with the points from our data set. We can see in both plots that there are a few outliers that get misclassified, but most of our Iris types cluster together in distinct groups.

The code to perform a logistic regression in scikit is similar to the code we used in the previous section to perform a linear regression. First, we need to create our training and test sets, then we fit a logistic regression.

To split the training and test sets, use the following code:

```
x_train = [ x for (i, x) in enumerate(iris.data) if i%10 !=0 ]
x_test = [x for (i, x) in enumerate(iris.data) if i%10 == 0]

y_train = [ y for (i, y) in enumerate(iris.target) if i%10 != 0 ]
y_test = [ y for (i, y) in enumerate(iris.target) if i%10 == 0 ]
```

For this tutorial, we split our data into blocks of 10 elements, putting the first element into the test set and the remaining nine elements into the training set. To ensure our data contains selections from all three classes, we'll need to use a more involved process in this example than previous examples.

Once we construct our training and test sets, fitting the logistic regression requires two lines of code. The second line of code uses our trained logistic regression to predict the Iris types of our test set.

```
logr = LogisticRegression()
logr.fit(x_train, y_train)
y_pred = logr.predict(x_test)
```

Part II: Building a Real-time Prediction Engine Using Redis 4.0.0 and the Redis-ML Module

As with our linear regression example, we can build a logistic regression predictor using Redis.

The Redis-ML module provides `ML.LOGREG.SET` and `ML.LOGREG.PREDICT` functions to create logistic regression keys.

To add a logistic regression model to Redis, you need to use the `ML.LOGREG.SET` command to add the key to the database.

The `ML.LOGREG.SET` command has the following form:

```
ML.LINREG.SET key intercept coeef [...]
```

The `ML.LOGREG.PREDICT` function is used to evaluate the logistic regression from the feature values and has the form:

```
ML.LOGREG.PREDICT key feature [...]
```

The order of the feature values in the `PREDICT` command must correspond to the coefficients. The result of the `PREDICT` command is the probability that an observation belongs to a particular class.

To use Redis to construct a multiclass classifier, we have to emulate the one-vs-the-rest (OvR) procedure (also known as one-vs-all (OvA)) used for multiclass classification. In the OvR procedure, multiple classifiers are created and each is used to determine the probability of an observation being in a particular class. The observation is then labeled with the class in which it is most likely to be a member.

For our three-class Iris problem, we need to create three separate classifiers, each determining the probability of a data point being in that particular class. The scikit `LogisticRegression` object defaults to OvR and fits the coefficients for three separate classifiers.

To emulate this procedure in Redis, we first create three logistic regression keys corresponding to the coefficients fit by scikit.

```
r = redis.StrictRedis('localhost', 6379)
for i in range(3):
      r.execute_command("ML.LOGREG.SET",
"iris-predictor:{}".format(i), logr.intercept_[i], *logr.coef_[i])
```

We emulate the OvR prediction procedure that takes place in the LogisticRegression.predict function by iterating over our three keys and taking the class with the highest probability. The following code executes the OvR procedure over our test data and stores the resulting labels in a vector.

```
# Run predictions in Redis
r_pred = np.full(len(x_test), -1, dtype=int)

for i, obs in enumerate(x_test):
       probs = np.zeros(3)
       for j in range(3):
                probs[j] = float(r.execute_command("ML.LOGREG.PREDICT", "iris-predictor:{}".format(j), *obs))
       r_pred[i] = probs.argmax()
```

Part III: Comparing Results

We compare the final classifications by printing out the three result vectors.

```
# Compare results as numerical vector
print("y_test = {}".format(np.array(y_test)))
print("y_pred = {}".format(y_pred))
print("r_pred = {}".format(r_pred))
```

The output vectors show the actual Iris species (y_test) and the predictions made by scikit (y_pred) and Redis (r_pred). Each vector stores the output as an ordered sequence of labels, encoded as integers. As you can see, Redis and scikit made identical predictions, including the mislabeling of one Virginica as a Versicolor.

```
y_test = [0 0 0 0 0 1 1 1 1 1 2 2 2 2 2]

y_pred = [0 0 0 0 0 1 1 2 1 1 2 2 2 2 2]

r_pred = [0 0 0 0 0 1 1 2 1 1 2 2 2 2 2]
```

While you likely do not have a need for an Iris classifier, you have learned how to use the Redis-ML module to implement a highly available, real-time classifier for your own data.

Matrix Operations

Matrices are used for a wide range of applications, from linear transforms to representing multivariate probability distributions. Because matrices are so common in machine learning, statistics, finance, and a host of other domains, they are a natural addition to Redis.

The Redis-ML module adds matrices as a native Redis data type. It also provides built-in mathematical operations that combine matrices to create new values.

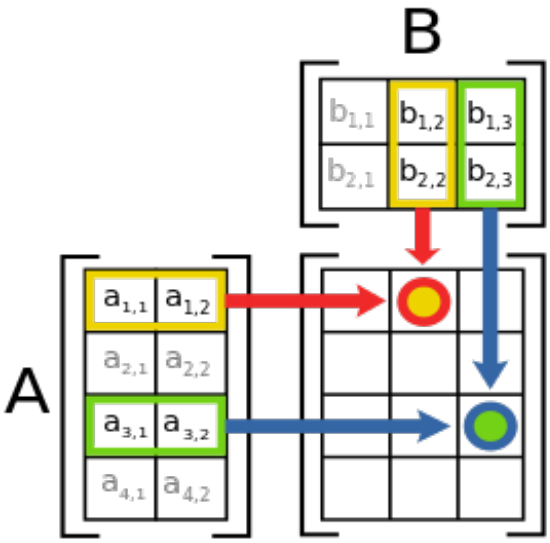

Figure 7. Schematic depiction of the matrix product AB of two matrices A and B. (Credit: File:Matrix multiplication diagram 2.svg. From Wikimedia Commons, the free media repository.)

Because matrices are so common in machine learning, statistics, finance, and a host of other domains, they were a natural addition to Redis.

The Redis-ML module adds matrices as a native Redis data type. It also provides built-in mathematical operations that combine matrices to create new values.

The reading and writing of matrix values is performed through the ML.MATRIX.SET and the ML.MATRIX.GET commands which have the following syntax:

```
ML.MATRIX.SET key n m entry11 .. entrynm
```

```
ML.MATRIX.GET key
```

When working with the Redis-ML module, remember that commands use row-major format (i.e. the consecutive elements of a row reside next to each other). Multiplication and addition are supported by the `ML.MATRIX.MULTIPLY` and the `ML.MATRIX.ADD` commands.

```
ML.MATRIX.MULTIPLY a b result
```

```
ML.MATRIX.ADD a b result
```

These commands combine two matrices that are already in Redis and store the result in a new key.

If we wanted to compute a basic Matrix equation such as *y = Ax + b* using the Redis-ML module, we would enter the following commands:

```
ML.MATRIX.SET a 3 3 4 0 0 0 2 0 0 0 1

ML.MATRIX.SET x 3 1 1 4 1

ML.MATRIX.SET b 3 1 1 1 1

ML.MATRIX.MULTIPLY a x tmp

ML.MATRIX.ADD tmp b y

ML.MATRIX.GET y
```

Redis returns the result to our client with the shape of the matrix (in this case three rows and one column) followed by each of the elements of the matrix in row-major order:

```
1) (integer) 3

2) (integer) 1

3) "5"

4) "9"

5) "2"
```

We could also compute the matrix equation A' = cA (where c is a scalar value) using the following code:

```
127.0.0.1:6379> ML.MATRIX.SET a 3 3 4 0 0 0 2 0 0 0 1
```

```
127.0.0.1:6379> ML.MATRIX.SCALE a 3
```

```
127.0.0.1:6379> ML.MATRIX.GET a
```

Redis returns:

```
 1) (integer) 3

 2) (integer) 3

 3) "12"

 4) "0"

 5) "0"

 6) "0"

 7) "6"

 8) "0"

 9) "0"
```

```
10) "0"

11) "3"
```

Decision Trees

Decision trees are a predictive model used for classification and regression problems in machine learning. They model a sequence of rules as a binary tree. The interior nodes of the tree represent a split (determined by a rule) and the leaves represent a classification or value.

Each rule in the tree operates on a single feature of the dataset. If the condition of the rule is met, the data point moves to the left; otherwise, it moves to the right. For categorical features (enumerations), the test the rule uses is membership in a particular category. For features with continuous values, the test is "less than" or "equal to."

To evaluate a data point, start at the root node and traverse the tree by evaluating the rules in the interior node, until a leaf node is reached. The leaf node is labeled with the decision that is returned.

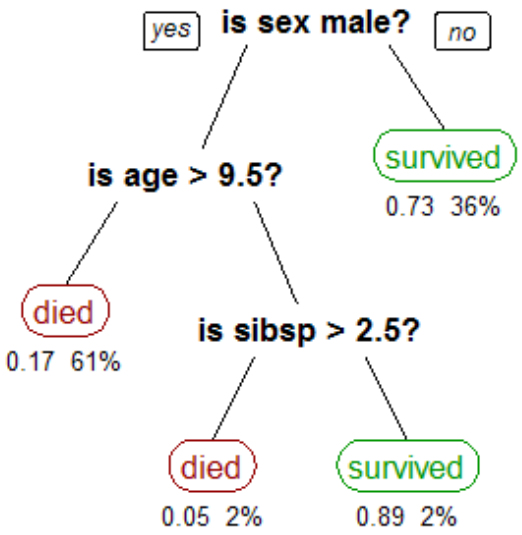

Figure 8. Sample decision tree that classifies survival rates of Titanic passengers (and probabilities for correct answers). (Credit: Algorithms and Ideas in Java blog.)

Many different algorithms (e.g. recursive partitioning, top-down induction) can be used to build decision trees, but the evaluation procedure is always the same. To improve the accuracy of decision trees, they are often aggregated into random forests which use multiple trees to classify a data point and then take the majority decision across the trees as a final classification.

Tutorial: Predicting the Survival of Titanic Passengers

In this tutorial, we will build a Titanic passenger survival predictor using the popular Python scikit-learn package and Redis.

> ***Note:*** *The sample code in this tutorial is written in Python and requires a Redis instance running Redis-ML. The instructions for setting up the appropriate Redis environment are identical for all tutorials in this white paper and can be found in the overview of the Redis-ML Features section.*

Part I: Readying the Data

We will be using a copy of the classic Titanic RMS dataset from the Vanderbilt archives. Available for download at http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.xls, this data set contains records for 1309 Titanic passengers. Each record consists of 14 fields:

- pclass (passenger class)
- survived (passenger survival)
- name
- sex
- age
- sibsp (number of siblings/spouses on board)
- (parch number of parents/children on board)
- ticket (ticket number)
- fare
- cabin (cabin number)
- embarked (port of embarkation)
- boat (life boat number, if survived)
- body (body number, if didn't survive and body found)
- home.dest (final destination)

A cursory scan of our dataset in Excel shows quite a bit missing data. These missing fields will negatively impact the accuracy of our results so we'll need to do some cleanup before building our decision tree. To do this, we'll use pandas (see http://pandas.pydata.org/getpandas.html), a Python data analysis library, to preprocess our data. You can install the pandas library using your preferred package manager or pip, the Python package manager (shown next):

```
pip install pandas
```

Using pandas, we can get a quick breakdown of the count of values for each of the record classes in our data:

```
pclass       1309

survived     1309

name         1309

sex          1309

age          1046

sibsp        1309

parch        1309

ticket       1309

fare         1308

cabin         295

embarked     1307

boat          486

body          121

home.dest     745
```

Since the *cabin*, *boat*, *body* and *home.dest* fields have a large number of missing values, we are simply going to drop them from our dataset. We're also going to drop the ticket and name fields, since they have little predictive value. That leaves our predictor with a feature set that includes eight fields. In addition, there are still several rows with missing data so, for simplicity, we will remove those passenger records from our dataset as well.

```
import pandas as pd

# load data from excel
orig_df = pd.read_excel('titanic3.xls', 'titanic3', index_col=None)

# remove columns we aren't going to work with, drop rows with missing data
df = orig_df.drop([“name”, "ticket", "body", "cabin", "boat", "home.dest"], axis=1)
df = df.dropna()
```

The final preprocessing we need to perform on our data is to encode categorical data using integer constants. The *pclass* and *survived* columns are already encoded as integer constants, but the *sex* and *embarked* columns use string values and letter codes respectively. The scikit package provides utilities in the preprocessing subpackage with which to transform non-integer encoded categorical features:

```
from sklearn import preprocessing

# convert enumerated columns (sex,)
encoder = preprocessing.LabelEncoder()
df.sex = encoder.fit_transform(df.sex)
df.embarked = encoder.fit_transform(df.embarked)
```

Part II: Building a Decision Tree

Now that we have cleaned our data, we can compute the mean value for several of our feature columns, grouped by passenger class and sex.

pclass	sex	survived	age	sibsp	parch	fare
1	female	0.961832	36.839695	0.564885	0.511450	112.485402
	male	0.350993	41.029250	0.403974	0.331126	74.818213
2	female	0.893204	27.499191	0.514563	0.669903	23.267395
	male	0.145570	30.815401	0.354430	0.208861	20.934335
3	female	.0473684	22.185307	0.736842	0.796053	14.655758
	male	0.169540	25.863027	0.488506	0.287356	12.103374

Notice the significant differences in survival rates between men and women based on passenger class. Our algorithm for building a decision tree will discover these statistical differences and use them to choose features to split on.

We will use scikit-learn to build a decision tree classifier over our data. We start by splitting our cleaned data into a training and test set. Using the following code, we'll split out the label column of our data (survived) from the feature set and reserve the last 20 records of our data for a test set.

```
X = df.drop(['survived'], axis=1).values
Y = df['survived'].values

X_train = X[:-20]
X_test  = X[-20:]
Y_train = Y[:-20]
Y_test  = Y[-20:]
```

Now we can create a decision tree with a maximum depth of 10.

```
# Create the real classifier depth=10
cl_tree = tree.DecisionTreeClassifier(max_depth=10, random_state=0)
cl_tree.fit(X_train, Y_train)
```

Our depth-10 decision tree is difficult to display in page format, so to help you visualize the structure of the decision tree, we've created a second tree and limited the tree's depth to three.

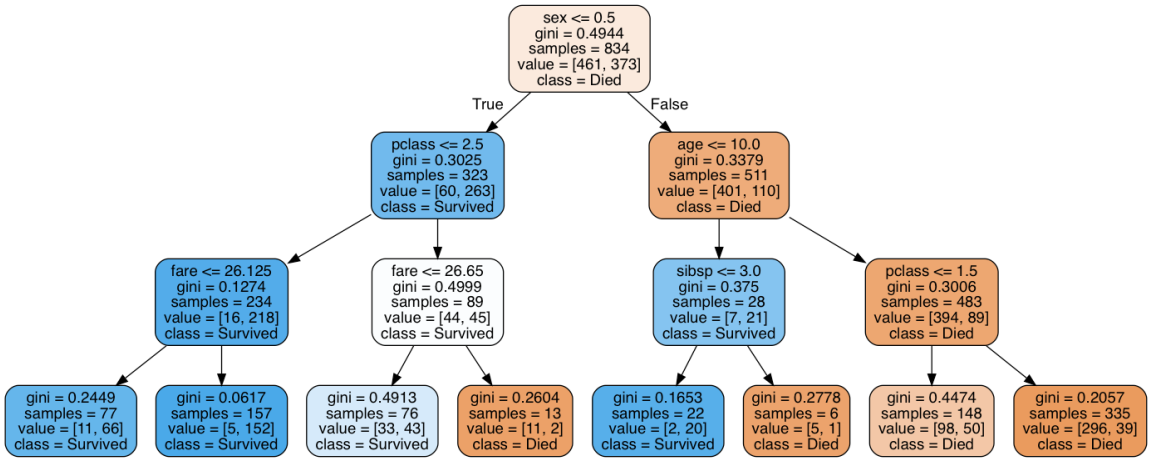

Figure 9. A depth-3 decision tree for Titanic survival rates, as learned by the classifier.

Part III: Loading the Redis Predictor

The Redis-ML module provides two commands for working with random forests:

- `ML.FOREST.ADD` creates a decision tree within the context of a forest
- `ML.FOREST.RUN` evaluates a data point using a random forest

The `ML.FOREST` commands have the following syntax:

```
ML.FOREST.ADD key tree path ((NUMERIC|CATEGORIC) attr val | LEAF val [STATS]) [...]

ML.FOREST.RUN key sample (CLASSIFICATION|REGRESSION)
```

Each decision tree in Redis-ML must be loaded using a single `ML.FOREST.ADD` command. The `ML.FOREST.ADD` command consists of a Redis key, followed by an integer tree id, followed by node specifications. Node specifications consist of a path, a sequence of . (root), l and r, representing the path to the node in a tree. Interior nodes are splitter or rule nodes and use either the NUMERIC or CATEGORIC keyword to specify the rule type, the attribute to test against and the value of threshold to split. For NUMERIC nodes, the attribute is tested against the threshold and if it is less than or equal to it, the left path is taken; otherwise the right path is taken. For CATEGORIC nodes, the test is equality. Equal values take the left path and unequal values take the right path.

The decision tree algorithm in scikit-learn treats categoric attributes as numeric, so when we represent the tree in Redis, we will only use NUMERIC node types. To load the scikit tree into Redis, we will need to implement a routine that traverses the tree. The following code performs a pre-order traversal of the scikit decision tree to generate a `ML.FOREST.ADD` command (since we only have a single tree, we generate a simple forest with only a single tree).

```
# scikit represents decision trees using a set of arrays,
# create references to make the arrays easy to access
the_tree = cl_tree
```

```
t_nodes = the_tree.tree_.node_count
t_left = the_tree.tree_.children_left
t_right = the_tree.tree_.children_right
t_feature = the_tree.tree_.feature
t_threshold = the_tree.tree_.threshold
t_value = the_tree.tree_.value
feature_names = df.drop(['survived'], axis=1).columns.values

# create a buffer to build up our command
forrest_cmd = StringIO()
forrest_cmd.write("ML.FOREST.ADD titanic:tree 0 ")

# Traverse the tree starting with the root and a path of “.”
stack = [ (0, ".") ]
while len(stack) > 0:
      node_id, path = stack.pop()

      # splitter node -- must have 2 children (pre-order traversal)
      if (t_left[node_id] != t_right[node_id]):
             stack.append((t_right[node_id], path + "r"))
             stack.append((t_left[node_id], path + "l"))

             cmd = "{} NUMERIC {} {} ".format(path, feature_names[t_feature[node_
      id]], t_threshold[node_id])
             forrest_cmd.write(cmd)
      else:
             cmd = "{} LEAF {} ".format(path, np.argmax(t_value[node_id]))
             forrest_cmd.write(cmd)

# execute command in Redis
r = redis.StrictRedis('localhost', 6379)
r.execute_command(forrest_cmd.getvalue())
```

With the decision tree loaded into Redis, we can create two vectors to compare the predictions of Redis with the predictions from scikit-learn.

```
# generate a vector of scikit-learn predictors
s_pred = cl_tree.predict(X_test)

# generate a vector of Redis predictions
r_pred = np.full(len(X_test), -1, dtype=int)
for i, x in enumerate(X_test):
      cmd = "ML.FOREST.RUN titanic:tree "
      # iterate over each feature in the test record to build up the
      # feature:value pairs
      for j, x_val in enumerate(x):
             cmd += "{}:{},".format(feature_names[j], x_val)
      cmd = cmd[:-1]
      r_pred[i] = int(r.execute_command(cmd))
```

To use the `ML.FOREST.RUN` command, we must generate a feature vector consisting of a list of comma-separated <feature>:<value> pairs. The <feature> portion of the vector is a string feature name that must correspond to the feature names used in the `ML.FOREST.ADD` command.

When comparing the r_pred and s_pred prediction values against the actual label values, you can see that Redis' predictions are identical to those of the scikit-learn package, including the misclassification of test items 0 and 14.

```
Y_test: [0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0]
r_pred: [1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
s_pred: [1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
```

Redis-ML Benefits

Across the landscape of existing machine learning toolkits, most focus on the act of training the model. The subsequent problem of putting a trained model into production and having it generate not only accurate predictions, but accurate predictions in real time without any performance tradeoffs, is typically left entirely to the developer. This is a tall order as it involves the time-consuming and operationally difficult task of building, testing, and implementing a custom service.

This is where Redis-ML plays a very important role. Once the machine learning models are trained and ready, Redis-ML makes it easy to implement recommendations or predictions for the application using simple APIs, without having to implement custom code or set up a highly available and scalable infrastructure to support the activity. Developers can build their predictive engines within the familiar, full-featured, and highly performant Redis data store, and piggyback off the many advantages Redis brings to the table, including:

- Built-in Data Management
- Language Interoperability
- Reduced Operational Overhead
- High Performance for Faster Prediction Generation
- Seamless Scalability

Built-in Data Management

Although Redis-ML predictive data structures such as the *linear regression object* or *random forests* are very different from the built-in Redis 4.0 data structures such as sets and hashes, it's important to realize that Redis-ML keys are still Redis keys. As a result, a Redis-ML key, like any Redis key, can be maintained using Redis' standard key management commands (see https://redis.io/commands#generic for key management command details).

Redis already provides developers with a managed keyspace for storing data. Additional statistical models can be added to an application with a simple SET command, allowing developers to maintain multiple versions of machine learning models for cases in which data needs to be reprocessed.

Redis-ML also avoid the layers of abstraction that encumber other databases. As with Redis data structures, machine learning data structures are served with their native "verbs" directly from memory, without requiring additional layers of translation.

Language Interoperability

Redis-ML provides interoperability with all programming languages including Scala, Node.js, .Net, Python, and others. This means your machine learning models are not restricted to use with applications that share the same language; they can be accessed concurrently by applications developed in different languages using Redis-ML's simple API.

Reduced Operational Overhead

Redis is already a part of most companies' technology stacks. Likely, your operations staff already understand how to scale, manage and monitor Redis instances—they may even have automated many deployment tasks. So employing Redis-ML for your machine learning needs requires considerably less overhead (in terms of time, resources, and expertise) than adding operational support for a custom, homegrown service that requires a new set of skills.

High Performance for Faster Prediction Generation

Redis maintains all data in memory, which makes it extremely fast. This extends to machine learning models as well, which, as with other Redis built-in data structures, are stored in their native formats. This allows you to avoid the need to generate them from file systems or other disk based data stores, a process which usually involves long serialization/deserialization overheads with slow disk accesses.

Storing and serving your trained models directly from in-memory Redis also parallelizes access to the models and significantly improves performance. In benchmarks, Redis has been measured as thirteen times faster than homegrown Java applications in predictive operations (see https://redis.com/modules/machine-learning for benchmark details).

Seamless Scalability

Delivering predictions with ever-evolving and -growing precision requires larger machine learning models. But existing solutions cannot hold the model in memory when it grows beyond the memory available in a single node. This triggers the serialization/deserialization to disk and immediately reduces performance.

The Redis-ML module, however, takes full advantage of Redis' in-memory distributed architecture to scale the database to any size needed in a fully automated manner. To scale up a Redis-based predictive engine, more Redis nodes are simply deployed and a replication topology with a single master node and multiple replica nodes is created. Updates to statistical models are then written to the master node and automatically distributed to the replicas—no additional code needs to be written (as would be with a custom application).

Conclusion

Redis-ML spans the divide between the training and ultimate deployment of machine learning models. This add-on Redis module combines highly specialized machine learning features such as linear regression, logistic regression, matrix operations, and decision trees with Redis' high performance and flexibility in order to accelerate the delivery of reliable, real-time predictive analytics.

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com