In professional golf and tennis tournaments, athletes welcome any competitive advantage. Looking beyond the current time horizon into the future allows players to position their rackets before the ball arrives and to select the appropriate club based on future weather conditions. The Predictive Cloud Computing project is no different. Time series algorithms project web origin traffic trends into the future so that cloud services can be provisioned or de-provisioned to anticipate sporting content demand. Sinusoidal patterns with a trend, cycle, and level that have been learned from historical data at rest are ensembled together with real-time demand patterns. The combination of InfoSphere Stream, InfoSphere BigInsights, RabbitMQ, WebSphere Java Application, Apache math commons, and custom algorithms have produced forecasts with an 18.44% mean absolute percentage error (MAPE). As a result, provisioning cloud services for future demand loads has saved thousands of compute minutes.

In this article, we provide an overview of the forecasting framework that enabled the PCC system to look beyond the current time horizon. We describe the fundamental concepts of time series forecasting and examine the data that was used throughout the system. We examine the architectures that processed data at rest from a previous year's sporting tournament and data in motion as logs were streamed into aggregation algorithms. Next, we discuss the development of our own custom forecasters that include standard regression, vector analysis, and numerical analysis of server log data. The section that follows depicts our development of several post-processors that smoothed and removed anomalous spikes. Finally, we show the overall accuracy of the forecasting system and its results. Throughout this article, we include numerous sample code listings as examples for developers.
“Throughout our tournament schedule, we saved up to 51% of computing hours or 134.5 hours of compute time within our 3-active clouds during a typical tournament day. At that rate, over the span of 107 competition days, the PCC could save up to 14,388 hours of computing resources.”

Fundamentals of time series forecasting

Time series is a fundamental technique to forecast data trends into the future. The most effective type of forecasting is typically incurred by regressing data that has a cyclical-based pattern. Autoregressive integrated moving average (ARIMA) is a model that is generally used throughout the forecasting field. Within our work, we have developed additional algorithms that implement advanced numerical analysis that complement traditional techniques such as ARIMA. As an example, here is an equation that shows a traditional cyclical pattern similar to our web traffic data when we filter out spike events:

$$f(t) = ax + bx + ct$$

This equation encodes important information trends for us to forecast future server use into the future. The $a$ variable represents the amplitude of the signal, which determines the maximum number of hits throughout our servers that we must maintain. The $b$ variable encodes the trend or the way the data is generally changing. In our problem space, we can determine the growth of the traffic or the slope. The variable $c$ lets us know the steady level of hits for our servers that serve tournament content. Finally, the trigonometric function $\sin(\theta)$ provides the seasonality of our data. The implementation of custom forecasters that project the amplitude, level, trend, and seasonality beyond the current time enabled the PCC to be proactive.

Before custom forecasters were implemented within PCC, serving microservices and a digital contextual user experience was reactive only. Figure 1 shows that the difference between server capacity and server load was not optimized and risked not having enough servers provisioned if any of the variables in the equation changed. A proactive model produced by the PCC looked 24 hours into the future and closed the gap between server capacity and server load. Overall, the system achieved a 10.25% mean absolute percentage error.

**Figure 1. The overall goal of the PCC was to proactively provision or de-provision cloud services.**

Throughout the Tennis Grand Slam tournaments, The Masters, and the United States Golf tournament, many different kinds of data were available to achieve high forecasting accuracy to continuously serve a global audience. Scores, schedule, injuries, weather, logs, tweets,
and simulation data provide the foundation to discover the signal for accurate forecasting and prediction.

Figure 2 presents the overall architecture of forecasting cyclical patterns and combining with a predictive model to include spikes in traffic. At step 1, real-time logs are streamed by the PCC pre-processors. The logs are accumulated by a Python program across all origin machines and pushed to RabbitMQ. An InfoSphere Streams process consumes the logs and aggregates the traffic counts each minute and sends the results to a Java application. The pre-processors impute any missing data, remove anomalous events that could be errors, and smooths the curves with a window average. The pre-processed data is sent to the time series ensemble for cyclical forecasting. A total of five complementary forecasters are applied and combined based on time horizon accuracy. The results of the cyclical forecasting are sent to post-processors in step 2. The post-processors time shift the data for a given time zone, smooth the forecast, and remove any spikes in forecast traffic. Continually, data such as scores, schedules, tweets, and web content are streamed to distributed feature-extractors that run on UIMA-AS, the same technology with which Watson won Jeopardy!

Figure 2. The forecasting and diverse data consumption during sporting events produced highly accurate predictions.

After we have finished the cyclical forecasting, the PCC simulates golf and tennis tournaments forward into time. Steps 4, 5, and 6 distribute simulation across several compute nodes to provide simulations near real time. A bank of feature extractors discovers 23 golf and 18 tennis predictors to estimate spikes in server traffic. For example, assume that, during golf, PCC determines if a featured group of golf players will be in play. If so, traffic will likely spike over the golf cloud services.

Next, in step 8, the output of the predictive model and cyclical forecasting are passed to a residual forecaster to correct for any mathematical errors. Both of the values of future demand are combined at step 9 with a half-life function. The half-life function is trained on the previous year's tournament to coerce the overall cyclical and spike forecast into a desired state to further boost accuracy. Finally, the PCC provisions or de-provisions cloud resources in preparation for cloud consumption.

Log data

Through each event, we use a content data network (CDN) to cache static content to protect origin servers. The number of edge requests reaches 1.9 billion per 24-hour period. About 6% of all traffic trickles down to the origin for dynamic content or when a time to live (TTL) expires. Of the 112.8 million requests to the origin, the magnitude of hits is not evenly distributed across time.
Figure 3. The number of edge hits through a full 24 hours of a typical tournament is in the range of 1.9 billion. About 5.6% of the traffic trickles down to origin servers.

<table>
<thead>
<tr>
<th>Requests</th>
<th>Number of requests</th>
<th>% of requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge requests</td>
<td>1.9 Billion</td>
<td>94.4%</td>
</tr>
<tr>
<td>Midday requests</td>
<td>199,469</td>
<td>0%</td>
</tr>
<tr>
<td>Origin requests</td>
<td>112.8 Million</td>
<td>5.6%</td>
</tr>
</tbody>
</table>

As shown in Figure 4, real-time logs from the origin are streamed through RabbitMQ to InfoSphere Streams and pushed to a WebSphere Java Application. The logs are parsed into counts per minute and saved into DB2. As a result, the forecasters described in Figure 2 can utilize the most recent information to project forecasts into the future.

Figure 4. Origin server logs are streamed through RabbitMQ to InfoSphere Streams and finally sent to a WebSphere Java application.

Data pre-processing

The log data used for forecasting can have many irregular spikes from bots or missing data from date and time rounding. Any of these cases can cause the cyclical pattern to be less accurate over a 24-hour period. Because the forecasting ensemble uses a predictive model for traffic spikes, a smooth general curve produces a gross estimation of tournament content demand.

Listing 1 depicts code that smooths out historical log information with a moving average window. The duration of the window can be parameterized so that elongated times further dampen general traffic spikes. Similar logic is used to impute any missing traffic magnitudes for a single minute throughout 1440 minutes in the day. If a data point is missing, an averaging window computes a nearest-neighbor value for insertion into the time series data.

Listing 1. Depicts the moving average processor within Java.

```java
class MovingAverageProcessor {
    private static final double DEFAULT_LENGTH = 20;
    private final double length;
```

Next, the system determines the seasonality of the data. The seasonality is coerced into a 24-hour period because the traffic trends day over day have a similar pattern. When the start and end time of a 24-hour period is determined, we stratify historical data in a similar pattern. As a result, we can determine an average curve or cyclical pattern that is used in later forecasters. However, we can also filter out any historical curve that deviates from all of the current data trends. This step is extremely important to remove historical data that might have missing data or incorrect logging information.

After all of the specified pre-processors within a JSON file have completed, the traffic trend is sent to an ensemble of forecasters.
Ensembling of historical forecasters

Several different types of historical forecasters complement each other to provide a diversity of techniques that provide highly accurate cyclical forecasting. The cohort of forecasters implement different time horizons such as short- (1-2 hours), mid- (10-15 hours), and long-term (24 hours), for independent evaluation before being combined together with a novel half-life weighting scheme. Figure 5 shows the historical forecasters within the context of the overall forecasting framework.

Figure 5. The overall forecasting architecture includes a cohort or ensemble of historical forecasters.

In Listing 2, the `CyclicalForecaster` class maintains a builder for each implemented historical forecaster. Generally, the PCC used partial average, adjusted average, quadratic, cubic, and vector forecasters throughout golf and tennis events. The adjusted average and partial average forecasters provided long-term forecasting through a statistical and shifted average of the training data discovered by the pre-processing stage. The average forecasters provide a baseline forecast that can be combined with the more precise mid- and short-term forecasters. The vector forecaster is a numerical analysis projection that uses the Runge-Kutta fourth-order (RK4) algorithm to project into the future five hours ahead of the current time horizon. Next, the quadratic and cubic
polynomial forecasters regress into the past to project two hours into the future. This forecaster adds additional precision to the baseline cyclical forecast. The output from the combination of all forecasters is a forecast cyclical trend of web server traffic.

**Listing 2. The cyclical forecasters were implemented within Java code.**

```java
public class CyclicalForecaster {
  private static final Logger logger = LoggerFactory.getLogger(CyclicalForecaster.class);
  private final TrainingFold fold;
  private final TournamentConfiguration tourneyConfig;
  private final ForecastingConfiguration forecastingConfig;

  private CyclicalForecaster(CyclicalForecasterBuilder builder) {
    this.fold = builder.fold;
    this.forecastingConfig = builder.forecastingConfig;
    this.tourneyConfig = builder.tourneyConfig;
  }

  public Iterable<ForecastInstant> createForecast() {
    HistoricalEnsembleForecast historicalForecast = new HistoricalEnsembleForecast(this.tourneyConfig,
                                                                                 forecastingConfig.getCompositeForecast().getCyclicalForecast());
    final Iterable<ForecastInstant> historical = historicalForecast.forecast(fold);
    logger.trace("Before Residual Forecast: ", historical);
    final Iterable<ForecastInstant> residualForecast = doResidualForecast(historical, this.tourneyConfig);
    logger.trace("After Residual Forecast: ", residualForecast);
    final Output = residualForecast;
    for (final Postprocessor p : getPostprocessors(fold)) {
      finalOutput = p.postprocess(finalOutput);
    }
    return finalOutput;
  }

  private List<Postprocessor> getPostprocessors(final TrainingFold fold) {
    final int zoneOffset = this.tourneyConfig.getSite().getTimezoneOffset();
    return Lists.transform(this.forecastingConfig.getCompositeForecast().getCyclicalForecast().getPostprocessors(), new
                          Function<PostprocessorValue, Postprocessor>() {
                            @Override
                            public Postprocessor apply(PostprocessorValue input) {
                              Map<String, Number> delegate = Maps.newHashMap();
                              delegate.put(POSTPROCESSOR.TIMEZONE_FOCUS, zoneOffset);
                              Parameters systemParams = new Parameters(delegate);
                              return input.getType().build(fold, input.getParameters(), systemParams);
                            }
                          });
  }
}
```

Each of the forecasters is instantiated with a builder. From a software engineering perspective, we are able to plug in new forecasters as they are developed very quickly. Listing 3 depicts the builder code for the cyclical forecasters.
Listing 3. Each of the forecasters was instantiated by a builder.

```java
public static class CyclicalForecasterBuilder {
    private final TournamentConfiguration tourneyConfig;
    private final ForecastingConfiguration forecastingConfig;
    private final TrainingFold fold;

    public CyclicalForecasterBuilder(TournamentConfiguration tourneyConfig, ForecastingConfiguration forecastingConfig, TrainingFold fold) {
        this.tourneyConfig = tourneyConfig;
        this.forecastingConfig = forecastingConfig;
        this.fold = fold;
    }

    public CyclicalForecaster build() {
        return new CyclicalForecaster(this);
    }
}
```

Partial average forecaster

The partial average forecaster forecasts several hours into the future. In order to project future trends, the forecaster learns from the last two hours of historical or training data for each day of current and past events. The forecaster creates a forecast curve that is the average all shifted training curves points based on the two-hour matching shift. Each of the points is time normalized with the current curve so that half of the averaged curve overlaps with the current trend. The portion that does not overlap with the curve becomes the forecast. Further, if the time prediction is within a weekend, only weekend day trends are utilized as training data. The same logic applies to weekdays where the training data is stratified to use only weekday trends if the current time falls during the week. Figure 6 shows a graphical representation of the forecaster that used three hours of historical data.

Figure 6. The partial average forecaster uses any number of periods or seasons to produce a forecast.

Adjusted average forecaster

The adjusted average and partial average forecasters are similar with the following two exceptions:

- The error function.
- The best curve match shifts each training curve between the current data and the entire previous periods.

If the forecast does not overlap the current trend, the entire average curve is the forecast. This case generally happens at the beginning of the seasonality trend.
The equation below shows how PCC finds the best shift point for a good curve match.

The point \((a, b, c)\) represents an encoding of the best point defined where \(a\) is the gain, \(b\) is the time shift, and \(c\) is the magnitude shift to compute the best curve match on the current data using Powell Optimization. The overall objective function for the Powell Optimization is defined whereby the sum of penalties is minimized. As a result, the current forecast is shifted by time \(b\), server demand \(a\), and gain \(c\). The implementation details within Java can be found in Listing 4 and Listing 5.

**Listing 4. A general adjusted average forecaster written in Java.**

```java
public class AdjustedAverageForecaster extends AverageForecaster implements Forecaster {
    public static final Duration FORECAST_HORIZON = Duration.standardDays(1);
    public AdjustedAverageForecaster(final Duration forecastHorizon, final Iterable<Iterable<ForecastInstant>> trainingForecasts, final double numStandardDev) {
        super(forecastHorizon, trainingForecasts, numStandardDev);
    }
    public AdjustedAverageForecaster(final Duration forecastHorizon, final Iterable<Iterable<ForecastInstant>> trainingForecasts) {
        super(forecastHorizon, trainingForecasts);
    }
    @Override
    protected Match<Iterable<ForecastInstant>> getCurveMatch(final Iterable<ForecastInstant> curve, CurveMatcher curveMatcher) {
        return curveMatcher.getBestMatch(curve);
    }
}
```

**Listing 5. The business of the average forecasters discovers future trends.**

```java
public abstract class AverageForecaster implements Forecaster {
    private static final double DEFAULT_NUMBER_OF_STANDARD_DEVIATIONS = 1.0;
    private static final Duration FORECAST_HORIZON = Duration.standardDays(1);
    private static final Logger logger = LoggerFactory.getLogger(AverageForecaster.class);
    private final Optional<Iterable<ForecastInstant>> weekendForecast;
    private final Iterable<ForecastInstant> forecast;
    private final Duration forecastHorizon;
    public AverageForecaster(final Duration forecastHorizon, final Iterable<Iterable<ForecastInstant>> trainingForecasts, final double numStandardDev) {
        logger.debug("Building Average Forecaster from training data.");
        forecast = ForecastReportInstants.shiftedAveragePlusStDev(trainingForecasts, numStandardDev);
        final Iterable<Iterable<ForecastInstant>> weekend = GroupingCode.filter(trainingForecasts,
            GroupingCode.WEEKEND);
        if (!Iterables.isEmpty(weekend)) {
            weekendForecast = Optional.of(ForecastReportInstants.shiftedAveragePlusStDev(weekend, numStandardDev));
        } else {
            weekendForecast = Optional.absent();
        }
        this.forecastHorizon = forecastHorizon;
    }
    public AverageForecaster(final Duration forecastHorizon, final Iterable<Iterable<ForecastInstant>> trainingForecasts) {
        this(forecastHorizon, trainingForecasts, DEFAULT_NUMBER_OF_STANDARD_DEVIATIONS);
    }
    @Override
    public Duration getForecastHorizon() {
    }
}
```
Vector forecaster

The vector forecaster uses the RK4 method shown in the equation below to predict server demand 5 hours into the future evening hours using a modeled vector plot from the log training data. The forecast accepts a set of aligned curves, a vector field density and a forecast length to produce the forecast. A Loess interpolator is applied to all training curves to produce a vector field. An RK4 method is applied to the vector field that yields the final forecast.

\[
\frac{dy}{dx} = \frac{y'}{x'} = \frac{\frac{dy}{dt}}{\frac{dx}{dt}}
\]

Listing 6 depicts code that implements the equation.

**Listing 6. RK4 Java code implemented with the vector forecast.**

```java
private Iterable<ForecastInstant> traceRK4(final Vector position, final Vector velocity, final double t, final DateTime start) {
    final FirstOrderDifferentialEquations equations = new FirstOrderDifferentialEquations() {
        @Override
        public int getDimension() {
            return 4;
        }

        @Override
        public void computeDerivatives(final double t, final double[] y, final double[] yDot) {
            Vector secondDerive = Vector.ZERO;
            try {
                secondDerive = trainedField.getComputedVector(trainedField.getXCoord(y[0]),
                        trainedField.getYCoord(y[2]));
            } catch (final IndexOutOfBoundsException e) {
                logger.trace("Error computing derivative", e);
            }
            yDot[0] = 1;// x1'
            yDot[1] = secondDerive.getX();// x2'
            yDot[2] = y[3];// y1'
            yDot[3] = secondDerive.getY();// y2'
        }
    };
```
final double[] positionAndVelocity = new double[] { position.getX(), velocity.getX(), position.getY(), velocity.getY() };

final RungeKuttaIntegrator rk = new ClassicalRungeKuttaIntegrator(size);
final List<ForecastInstant> data = Lists.newArrayList();
rk.addStepHandler(new StepHandler() {
    @Override
    public void init(final double t0, final double[] y0, final double t) {
        // empty
    }
    @Override
    public void handleStep(final StepInterpolator interpolator, final boolean isLast) {
        final double[] d = interpolator.getInterpolatedState();
        data.add(new ForecastInstant(start.plusMillis((int) d[0]), d[2]));
    }
});
rk.integrate(equations, 0, positionAndVelocity, t, positionAndVelocity);
Collections.sort(data);
return data;

Figure 7 shows an example of aligned curves from three periods of data. The aligned curves have been shifted through the Powell Optimization process. The dots on the aligned curves produce the vector field whereby the RK4 method creates the forecast.

**Figure 7. The diagram shows the inner workings of the vector forecaster.**

![Diagram showing vector forecaster](image)

**Quadratic and cubic forecaster**

Any type of polynomial can be fit or regressed onto historical data. However, the PCC uses a quadratic and cubic form because any other polynomial of higher degree over-fits the data. Both of the forecasters regress onto historical data any parameterized time period into the past. The resulting models include a quadratic equation \( f(x) = ax^2 + bx + c \) and a cubic model \( f(x) = ax^3 + bx^2 + cx + d \).

The input of a future time \( t \) into the models yields a server traffic forecast value.

Listing 7 shows the use of a generic `PolynomialForecaster` class and implementation of a Forecaster to define the `CubicForeaster` class. The Apache Commons Math libraries provided the underlying modeling algorithms.

**Listing 7. The implementation of the cubic forecaster was in Java.**

```java
public class CubicForecaster extends PolynomialForecaster implements Forecaster {
```
Residual forecaster

The residual forecaster takes into account residual interpolation error from our math libraries. The forecasters match the amplitudes of historical demand to future forecast demand over a 24-hour period. The resulting residual curve is the maximum curve determined by the percentage error between the actual demand value and the interpolated values. The final forecast trend is multiplied by the residual curve to adjust for the residual interpolation error. Figure 8 provides a summary of the algorithm.

Figure 8. The general logic of the residual forecaster accounted for mathematical interpolation errors.

Ensemble of forecasters

As was shown in Figure 5, the historical or cyclical cohort contains multiple time series forecasts. Each of the forecasters has a complementary time horizon for forecasting. The further out into the future, the more uncertainty each forecaster introduces into the forecast. However, each forecaster’s accuracy is different at varying time periods. For example, the vector and polynomial forecasters should have more weight than the adjusted and partial forecasters when the time horizon is closer to the current time period but less weight as the forecast extends into the future.
A normalized half-life weight is applied to each individual forecast from every forecaster within the cohort to lessen uncertainty. The equation below depicts the standard half-life notation where \( N \) is the original demand, \( t \) is the time unit, and \( \alpha \) is a half-life estimate.

\[
N(t) = N_0 e^{-t/\alpha}
\]

The next equation calculates a forecast weight \( N(t) \) for forecaster \( f \) at time \( t \), given the number of minutes \( m \), that have passed. To lessen the degree of weight decay, "2" is substituted for the \( e \) (in the equation above) and normalizes the starting weight value to 1. Finally, the half-life value for each forecaster is half of the maximum number of minutes in the forecast.

The weights for each of the forecasters are normalized so that the sum of all forecaster weights at a particular forecast time equals to 1. As a result, each forecaster will contribute to the overall forecast of a minute given a weight.

Listing 8 shows some of the Java code for calculating the weight of each forecaster and then producing a final forecast value at a given time unit.

**Listing 8. The Java code depicts our implementation of the forecast weight normalization process.**

double weight;

\[
\text{if (relativeMinutes > 0)} \{ \\
\qquad \text{// if the forecast horizon is a "filler" or guarantees 24 hours, we want to find what is actually being forecast from the standard day minutes and then divide} \\
\qquad \text{// by 2 to find the half point.} \\
\qquad \text{if (forecaster.getForecastHorizon().getStandardMinutes() ==} \\
\qquad \text{Duration.standardDays(1).getStandardMinutes())} \\
\qquad \text{weight = Math.pow(2, -relativeMinutes / ((Duration.standardDays(1).getStandardMinutes() -} \\
\qquad \text{Minutes.minutesBetween(start, now).getMinutes()) / 2) * 1.1));} \\
\quad \text{else} \{ \\
\qquad \text{// the mid point of the forecast horizon is used as the mid point} \\
\qquad \text{weight = Math.pow(2, -relativeMinutes / (forecaster.getForecastHorizon().getStandardMinutes() / 2) *} \\
\qquad \text{1.1);} \\
\quad \} \\
\text{else \{} \\
\qquad \text{weight = 1;} \\
\text{\}} \\
\text{\} // all weights for a particular minute from all forecasters is summed to adjust the sum of weights to 1} \\
\text{sumWeights += weight;} \\
\text{\} // store each weight} \\
\text{weights.add(weight);} \\
\text{\} // store each forecast value} \\
\text{forecastScores.add(nearestInstant.getValue());} \\
\]

\[
\text{int parabolaMidX = Minutes.minutesBetween(new} \\
\text{DateTime(start.withZone(DateTimeZone.forOffsetHours(config.getTimezoneOffset()))).withTimeAtStartOfDay(),} \\
\text{now).getMinutes());} \\
\text{double weight;} \\
\text{// check to see if this is a forecast, otherwise, this is a historical trend} \\
\text{// the overall equation is the half: N(t) = No * e ^ (-t/T); No = original value, t = amount of time that has passed, T=the average half life} \\
\text{// for our purposes, No=1 since we start with a weight of 1. Also, e decays too quickly so we use 2.} \\
\text{// Our equation reduces, 2^((-m/h)^alpha); m=relative minutes that has passed, h=half life or mid point of our time horizon, alpha helps to ensure that we do not} \\
\text{// have weights of zero.} \\
\text{// if (relativeMinutes > 0) \{} \\
\text{// if the forecast horizon is a "filler" or guarantees 24 hours, we want to find what is actually being forecast from the standard day minutes and then divide} \\
\text{// by 2 to find the half point.} \\
\text{if (forecaster.getForecastHorizon().getStandardMinutes() ==} \\
\text{Duration.standardDays(1).getStandardMinutes())} \\
\text{weight = Math.pow(2, -relativeMinutes / ((Duration.standardDays(1).getStandardMinutes() -} \\
\text{Minutes.minutesBetween(start, now).getMinutes()) / 2) * 1.1));} \\
\text{else \{} \\
\text{// the mid point of the forecast horizon is used as the mid point} \\
\text{weight = Math.pow(2, -relativeMinutes / (forecaster.getForecastHorizon().getStandardMinutes() / 2) *} \\
\text{1.1);} \\
\text{\}} \\
\text{\} // all weights for a particular minute from all forecasters is summed to adjust the sum of weights to 1} \\
\text{sumWeights += weight;} \\
\text{// store each weight} \\
\text{weights.add(weight);} \\
\text{// store each forecast value} \\
\text{forecastScores.add(nearestInstant.getValue());} \\
\]
new DateTime(maxTime.withZone(DateTimeZone.forOffsetHours(config.getSite().getTimezoneOffset()))).getMinutes();
if (parabolaMidX < 0) {
    parabolaMidX += Days.ONE.toStandardMinutes().getMinutes();
} else if (parabolaMidX > Days.ONE.toStandardMinutes().getMinutes()) {
    parabolaMidX -= Days.ONE.toStandardMinutes().getMinutes();
}
final double parabolicScalar = (watson.getParabola().getSlope() * Math.pow(minuteOffset - parabolaMidX, 2)) + watson.getParabola().getPeak();
final double modWatsonBooster = watson.getBooster() * parabolicScalar;
if (modWatsonBooster * interpolation > ProvisionInstants.REQUESTS_PER_SERVER) {
    adjustedForecast = modWatsonBooster * interpolation;
    // use the maximum of cyclic or event -- maximum method
    final double currentInterpolation = interpolation(currentInterpolator, current, instant);
    double yValue = Math.max(adjustedForecast, currentInterpolation);
    combinedForecasts.add(instant.setValue(yValue));
} else {
    skip++;
    combinedForecasts.add(instant);
}

Data post-processing

The time series post-processors provide smoothing effects and reduce the impact of erroneous traffic, Twitter, scores, simulation, player, web semantic data, and schedule data, while following an aggressive provisioning policy. The forecasting post-processors are implemented after both the historical forecasters and the spike forecaster; however, within this article, we are limiting our discussion to the time after the historical forecasters only.

The final combined historical forecast through the half-life weighting algorithm can produce noisy and rigid forecasts. To prepare the cyclical pattern to be combined with the spike forecast, several post-processors boost accuracy and complementary properties for the spike forecaster. During our professional golf and tennis events, ground truth data or real-time data might be missing due to messaging lag. As a result, an imputer that was similar in the pre-processing fills in data that has been lost. In a few cases, the forecasters produce flat curves or repeating magnitudes, which generally does not happen with our server traffic. A flat-line post-processor removes long line segments by averaging corresponding historical data. Furthermore, because the PCC runs during events around the world, a time shift post-processor changes the time zone of the forecast curve.

Cloud provisioning

Servers are autonomically provisioned based upon the forecast demand output from the ensemble of forecasters. For each minute of the forecast, the number of servers to provision is calculated. The PCC always provisions at least one server if the forecast is so low that none would be otherwise provisioned. The equation below, when implemented in Java, determines how many cloud resources to provision or de-provision.

The number of servers to provision at time $t$ is represented by $p_{vt}$. The total capacity of the server, which is hardware dependent, is represented by $p$. Through experimentation and given our
tolerance for risk, we mitigated forecasting error with a constant buffer zone with \( \rho \). The value of the buffer zone was determined from historical event trends and year-over-year growth.

\[
P = \rho \cdot \frac{\sum_{i=1}^{n} f_i(x_i)}{n}
\]

The provisioning or de-provisioning of cloud services occurs in real time throughout each of the golf and tennis tournaments.

## Results

Throughout our tournament schedule, we saved up to 51% of computing hours or 134.5 hours of compute time within our 3-active cloud during a typical tournament day. At that rate, over the span of 107 competition days, the PCC could save up to 14,388 hours of computing resources. Further, Table 1 shows the accuracy of the PCC. The cyclical-based-only forecasters produced a mean absolute percentage error (MAPE) of 18.44%. The code for calculating MAPE is shown in Listing 9.

In the next article in this series, you will discover that we further reduced error by more than 8% by adding in the spike forecaster. The high accuracy of forecasting and the reduction of compute-hours waste certainly outweighed any risk of under-provisioning cloud services.

### Table 1. The accuracy measure, MAPE, enabled us to measure the performance of the forecast.

<table>
<thead>
<tr>
<th>Forecast Type</th>
<th>Mean Absolute Percentage Error (MAPE)</th>
<th>Mean Absolute Error (HPM)</th>
<th>Average Error (HPM)</th>
<th>Root Mean Squared Error (RMSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclical Based</td>
<td>18.44%</td>
<td>7,968</td>
<td>7,209</td>
<td>14,608</td>
</tr>
<tr>
<td>Cyclical and Event</td>
<td>10.25%</td>
<td>3,256</td>
<td>1,528</td>
<td>4,993</td>
</tr>
</tbody>
</table>

### Listing 9. The MAPE measure as implemented in Java.

```java
public class MeanAbsolutePercentageError implements ErrorMeasure {
    private final ErrorConditionMeasure condition;

    public MeanAbsolutePercentageError(ErrorConditionMeasure condition) {
        super();
        this.condition = condition;
    }

    @Override
    public double calcError(ImmutableList<ForecastInstant> groundTruth, ImmutableList<ForecastInstant> forecast)
            throws ErrorMeasureException {
        condition.canCalc(groundTruth, forecast);
        double delta = 0;
        for (int errorLoop = 0; errorLoop < Iterables.size(groundTruth); errorLoop++) {
            delta += Math.abs((Iterables.get(groundTruth, errorLoop).getValue() - Iterables.get(forecast,
                errorLoop).getValue()) / Iterables.get(groundTruth, errorLoop).getValue());
        }
        return (delta / Iterables.size(groundTruth)) * 100;
    }
}
```
To visualize the data through the PCC forecasting, several dashboards were available for webmasters, clients, developers, managers, and consultants. Figure 9 depicts a real-time visualization of the Predicted Cloud Computing forecast. The yellow line on the right is the time series component, while the red line to its left is the event-based component. The blue-boxed line is the server capacity. The goal is to never have traffic go beyond the serving capacity of the cloud. Otherwise, end users would receive resource errors and could not access portions of the sporting website. As time progressed, the vertical line would progress forward. Any graphs to the right of the vertical line are forecasts.

Figure 9. The overall forecast was displayed on a real-time dashboard.

The overall tournament dashboard displayed in Figure 10 provides a tournament summary. Several spark lines show the velocity of tweets about a specific tournament as well as the log accesses across all of our origin machines. The distributed chain simulator produces a predicted schedule of play that is displayed under the spark lines. A summarized forecast trend is placed below the simulated tournament. Other views show player popularity, web traffic trend, content crawl results, predictor contributions to the forecast, and general system message trends.

Figure 10. The overall dashboard enables users to view the different types of data and the results of the PCC.

Conclusion

In this article, we have shown how the PCC used a custom forecasting framework to determine how many cloud resources to provision or de-provision for future workloads. IBM InfoSphere Streams and RabbitMQ enabled the PCC to process and accumulate logs over one-minute
intervals as a sporting event progressed in real time. Large volumes of logs were stored within the Hadoop Distributed File System (HDFS) through IBM InfoSphere BigInsights and aggregated over a specified time block. The input into the forecasting framework was pre-processed to ensure each time block was uniform and consistent while the output was post-processed to further boost the system’s accuracy. Custom code and the Apache Math Common libraries supported the mathematical operations within our algorithms. Several equations, code blocks, and evaluation metric tables depict tangible implementation details.

In part 9 of the series, we will discuss the distributed feature extraction and predictive model used throughout the PCC system that complemented the sinusoidal forecasting system.
Related topics

- Forecasting configuration [video]
- Predictive Cloud Computing Efficiently Manages Digital Experiences and Showcases IBM Technology at Professional Golf and Tennis Tournaments [video]

© Copyright IBM Corporation 2016
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)