Bid by Weather



Demand for certain products and services varies greatly depending on the weather. For example, users are much more likely to search for information on amusement parks on a hot, sunny day than if it's cold and raining. An amusement park company may want to increase their bids when the weather is nice, but doing so every day would require a lot of manual work. With AdWords scripts, however, it's possible to programmatically fetch weather information and adjust bids in a matter of minutes.

This script uses Google Spreadsheets to store the list of campaigns and their associated locations. A call to the [OpenWeatherMap API](http://openweathermap.org/) is made for each location and weather conditions are calculated using some basic rules. If a rule evaluates to true, then a corresponding location bid multiplier is applied to the location targeting for the campaign.

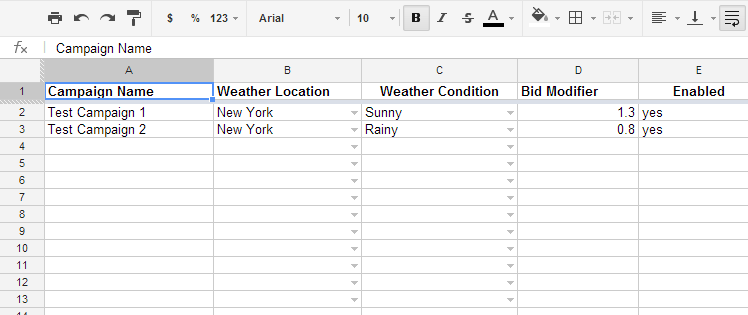
How it works

The script works by reading data off a spreadsheet. The spreadsheet consists of three individual sheets:

**1. Campaign data**

A set of rules determines the bid modifiers to be applied to campaigns when a weather condition is met. Required columns in this sheet are:

* Campaign name: The name of the campaign to be modified.
* Weather location: The location for which weather conditions should be checked.
* Weather condition: The weather condition for which this rule should be applied.
* Bid modifier: The location bid modifier to be applied if the weather condition is met.
* Enabled: Specify yesto enable a rule and no to disable it.



The sheet shown above has two rules:

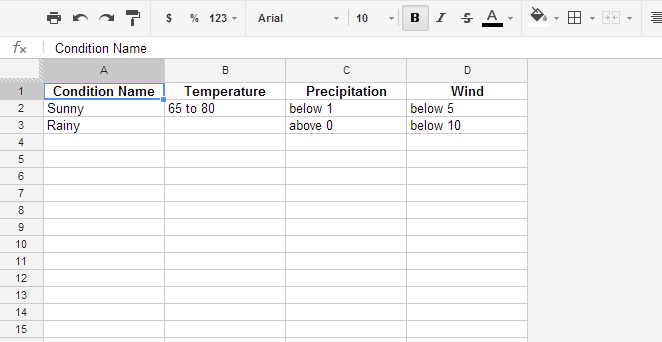
1. Apply a bid modifier of 1.3 to Test Campaign 1 if the weather in New York is Sunny.
2. Apply a bid modifier of 0.8 to Test Campaign 2 if the weather in New York is Rainy.

**Note:** For the rule to work, the specified campaign should already have a geo target for the specified weather location; the script won't modify existing geo targets of a campaign. Also, if multiple rules match for a given campaign and location, the last matching rule wins.

**2. Weather data**

This sheet defines the weather conditions used in the Campaign data sheet. Required columns are:

* Condition Name: The weather condition name (e.g. Sunny).
* Temperature: The temperature in Fahrenheit.
* Precipitation: The rain, in millimeters during the last 3 hours.
* Wind: The wind speed, in miles per hour.



The sheet shown above defines two weather conditions:

1. Sunny: Temperature is between 65 and 80 degrees Fahrenheit, precipitation below 1mm of rain in the last three hours, and wind speed less than 5 mph.
2. Rainy: Precipitation is above 0mm of rain in the last three hours and wind speed less than 10 mph.

Defining weather conditions

When defining weather conditions, you may specify the values as follows:

* below x: The specified value is below x (e.g. below 10)
* above x: The specified value is above x (e.g. above 70)
* x to y: The specified value is between x and y, inclusive (e.g. 65 to 80)

If you leave a cell blank, then the values of that parameter is not considered for calculation. For example, in the sheet shown above, the "Rainy" weather condition has an empty temperature column, so the temperature won't be considered when calculating this weather condition.

The weather conditions are ANDed together when calculating the weather condition. For this example, the "Sunny" weather condition is evaluated as:

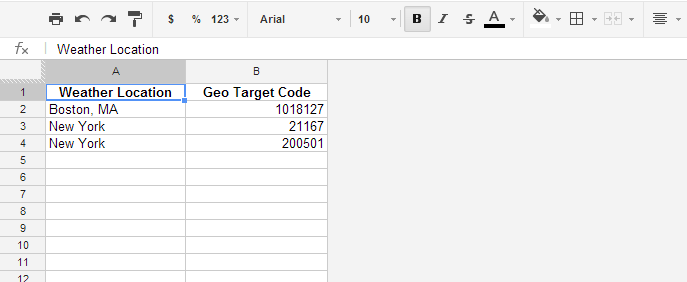
var isSunny = (temperature >= 65 && temperature <= 80) && (precipitation < 1) && (wind < 5);

**3. Weather location data**

This sheet defines the weather locations used in the Campaign data sheet, and consists of only two columns:

* Weather Location: This is a weather location name, as understood by the OpenWeatherMap API. You can refer to<http://openweathermap.org/help/city_list.txt> for a complete list of supported weather locations.
* Geo target code: This is a geo target code, as understood by AdWords scripts. You can refer to<https://developers.google.com/adwords/api/docs/appendix/geotargeting> for a list of supported geo target codes.

The script allows you to specify multiple geo target codes for a single weather location. This is supported because weather locations are not always as granular as the targeting options available in AdWords, and you may find it necessary to map a single weather location to multiple geo locations. Mapping single weather location to multiple geo locations can be done by having multiple rows with the same weather location but different geo codes for each row.



In the spreadsheet above, there are two weather locations defined:

1. Boston: Geo code 10108127
2. New York: Geo codes 21167, 20050

**Script logic**

The script starts by reading rules from all three sheets. It then attempts to execute each rule from the Campaign sheet in sequence.

For each executed rule, the script checks if the campaign targets the specified location. If it does, then the script retrieves the current bid modifier. Next, the weather conditions for that location is retrieved by making a call to OpenWeatherMap API. The weather condition rules are then evaluated to see if the weather condition for the location matches what is specified in the rule. If it does, and the new bid modifier is different from the current bid modifier, the script modifies the bid modifier for that location.

No changes are made if the weather condition doesn't match, if the bid modifier values are the same, or if the campaign doesn't target the location specified in the rule.

Setup

* Register for an API key at <http://openweathermap.org/appid>.
* Make a copy of the [template spreadsheet](https://goo.gl/SNE5H7) and edit your campaign and weather rules.
* Create a new script with the source code below.
* Update the OPEN\_WEATHER\_MAP\_API\_KEY and SPREADSHEET\_URL variables in the script.
* Schedule to run as required.

[Creating an AdWords script](https://developers.google.com/adwords/scripts/docs/solutions/bid-by-weather)

Source code

// Register for an API key at http://openweathermap.org/appid  
// and enter the key below.  
var OPEN\_WEATHER\_MAP\_API\_KEY = "INSERT\_OPEN\_WEATHER\_MAP\_API\_KEY\_HERE";  
  
// Create a copy of http://goo.gl/SNE5H7 and enter the URL below.  
var SPREADSHEET\_URL = 'INSERT\_SPREADSHEET\_URL\_HERE';  
  
// A cache to store the weather for locations already lookedup earlier.  
var WEATHER\_LOOKUP\_CACHE = {};  
  
  
/\*\*  
 \* The code to execute when running the script.  
 \*/  
function main() {  
  // Load data from spreadsheet.  
  var spreadsheet = SpreadsheetApp.openByUrl(SPREADSHEET\_URL);  
  var campaignRuleData = getSheetData(spreadsheet, 1);  
  var weatherConditionData = getSheetData(spreadsheet, 2);  
  var geoMappingData = getSheetData(spreadsheet, 3);  
  
  // Convert the data into dictionaries for convenient usage.  
  var campaignMapping = buildCampaignRulesMapping(campaignRuleData);  
  var weatherConditionMapping =  
      buildWeatherConditionMapping(weatherConditionData);  
  var locationMapping = buildLocationMapping(geoMappingData);  
  
  // Apply the rules.  
  for (var campaignName in campaignMapping) {  
    applyRulesForCampaign(campaignName, campaignMapping[campaignName],  
        locationMapping, weatherConditionMapping);  
  }  
}  
  
/\*\*  
 \* Retrieves the data for a worksheet.  
 \* @param {Object} spreadsheet The spreadsheet.  
 \* @param {number} sheetIndex The sheet index.  
 \* @return {Array} The data as a two dimensional array.  
 \*/  
function getSheetData(spreadsheet, sheetIndex) {  
  var sheet = spreadsheet.getSheets()[sheetIndex];  
  var range =  
      sheet.getRange(2, 1, sheet.getLastRow() - 1, sheet.getLastColumn());  
  return range.getValues();  
}  
  
/\*\*  
 \* Builds a mapping between the list of campaigns and the rules  
 \* being applied to them.  
 \* @param {Array} campaignRulesData The campaign rules data, from the  
 \*     spreadsheet.  
 \* @return {Object.<string, Array.<Object>> } A map, with key as campaign name,  
 \*     and value as an array of rules that apply to this campaign.  
 \*/  
function buildCampaignRulesMapping(campaignRulesData) {  
  var campaignMapping = {};  
  for (var i = 0; i < campaignRulesData.length; i++) {  
    // Skip rule if not enabled.  
  
    if (campaignRulesData[i][4].toLowerCase() == 'yes') {  
      var campaignName = campaignRulesData[i][0];  
      var campaignRules = campaignMapping[campaignName] || [];  
      campaignRules.push({  
          'name': campaignName,  
  
          // location for which this rule applies.  
          'location': campaignRulesData[i][1],  
  
          // the weather condition (e.g. Sunny)  
          'condition': campaignRulesData[i][2],  
  
          // bid modifier to be applied.  
          'bidModifier': campaignRulesData[i][3]  
      });  
      campaignMapping[campaignName] = campaignRules;  
    }  
  }  
  Logger.log('Campaign Mapping: %s', campaignMapping);  
  return campaignMapping;  
}  
  
/\*\*  
 \* Builds a mapping between a weather condition name (e.g. Sunny) and the rules  
 \* that correspond to that weather condition.  
 \* @param {Array} weatherConditionData The weather condition data from the  
 \*      spreadsheet.  
 \* @return {Object.<string, Array.<Object>>} A map, with key as a weather  
 \*     condition name, and value as the set of rules corresponding to that  
 \*     weather condition.  
 \*/  
function buildWeatherConditionMapping(weatherConditionData) {  
  var weatherConditionMapping = {};  
  
  for (var i = 0; i < weatherConditionData.length; i++) {  
    var weatherConditionName = weatherConditionData[i][0];  
    weatherConditionMapping[weatherConditionName] = {  
      // Condition name (e.g. Sunny)  
      'condition': weatherConditionName,  
  
      // Temperature (e.g. 50 to 70)  
      'temperature': weatherConditionData[i][1],  
  
      // Precipitation (e.g. below 70)  
      'precipitation': weatherConditionData[i][2],  
  
      // Wind speed (e.g. above 5)  
      'wind': weatherConditionData[i][3]  
    };  
  }  
  Logger.log('Weather condition mapping: %s', weatherConditionMapping);  
  return weatherConditionMapping;  
}  
  
/\*\*  
 \* Builds a mapping between a location name (as understood by wunderground API)  
 \* and a list of geo codes as identified by AdWords scripts.  
 \* @param {Array} geoTargetData The geo target data from the spreadsheet.  
 \* @return {Object.<string, Array.<Object>>} A map, with key as a locaton name,  
 \*     and value as an array of geo codes that correspond to that location  
 \*     name.  
 \*/  
function buildLocationMapping(geoTargetData) {  
  var locationMapping = {};  
  for (var i = 0; i < geoTargetData.length; i++) {  
    var locationName = geoTargetData[i][0];  
    var locationDetails = locationMapping[locationName] || {  
      'geoCodes': []      // List of geo codes understood by AdWords scripts.  
    };  
  
    locationDetails.geoCodes.push(geoTargetData[i][1]);  
    locationMapping[locationName] = locationDetails;  
  }  
  Logger.log('Location Mapping: %s', locationMapping);  
  return locationMapping;  
}  
  
/\*\*  
 \* Applies rules to a campaign.  
 \* @param {string} campaignName The name of the campaign.  
 \* @param {Object} campaignRules The details of the campaign. See  
\*      buildCampaignMapping for details.  
 \* @param {Object} locationMapping Mapping between a location name (as  
 \*     understood by wunderground API) and a list of geo codes as  
 \*     identified by AdWords scripts. See buildLocationMapping for details.  
 \* @param {Object} weatherConditionMapping Mapping between a weather condition  
 \*     name (e.g. Sunny) and the rules that correspond to that weather  
 \*     condition. See buildWeatherConditionMapping for details.  
 \*/  
function applyRulesForCampaign(campaignName, campaignRules, locationMapping,  
                               weatherConditionMapping) {  
  for (var i = 0; i < campaignRules.length; i++) {  
    var bidModifier = 1;  
    var campaignRule = campaignRules[i];  
  
    // Get the weather for the required location.  
    var locationDetails = locationMapping[campaignRule.location];  
    var weather = getWeather(campaignRule.location);  
    Logger.log('Weather for %s: %s', locationDetails, weather);  
  
    // Get the weather rules to be checked.  
    var weatherConditionName = campaignRule.condition;  
    var weatherConditionRules = weatherConditionMapping[weatherConditionName];  
  
    // Evaluate the weather rules.  
    if (evaluateWeatherRules(weatherConditionRules, weather)) {  
      Logger.log('Matching Rule found: Campaign Name = %s, location = %s, ' +  
          'weatherName = %s,weatherRules = %s, noticed weather = %s.',  
          campaignRule.name, campaignRule.location,  
          weatherConditionName, weatherConditionRules, weather);  
      bidModifier = campaignRule.bidModifier;  
      adjustBids(campaignName, locationDetails.geoCodes, bidModifier);  
    }  
  }  
  return;  
}  
  
/\*\*  
 \* Converts a temperature value from kelvin to fahrenheit.  
 \* @param {number} kelvin The temperature in Kelvin scale.  
 \* @return {number} The temperature in Fahrenheit scale.  
 \*/  
function toFahrenheit(kelvin) {  
  return (kelvin - 273.15) \* 1.8 + 32;  
}  
  
/\*\*  
 \* Evaluates the weather rules.  
 \* @param {Object} weatherRules The weather rules to be evaluated.  
 \* @param {Object.<string, string>} weather The actual weather.  
 \* @return {boolean} True if the rule matches current weather conditions,  
 \*     False otherwise.  
 \*/  
function evaluateWeatherRules(weatherRules, weather) {  
  // See http://bugs.openweathermap.org/projects/api/wiki/Weather\_Data  
  // for values returned by OpenWeatherMap API.  
  var precipitation = 0;  
  if (weather.rain && weather.rain['3h']) {  
    precipitation = weather.rain['3h'];  
  }  
  var temperature = toFahrenheit(weather.main.temp);  
  var windspeed = weather.wind.speed;  
  
  return evaluateMatchRules(weatherRules.temperature, temperature) &&  
      evaluateMatchRules(weatherRules.precipitation, precipitation) &&  
      evaluateMatchRules(weatherRules.wind, windspeed);  
}  
  
/\*\*  
 \* Evaluates a condition for a value against a set of known evaluation rules.  
 \* @param {string} condition The condition to be checked.  
 \* @param {Object} value The value to be checked.  
 \* @return {boolean} True if an evaluation rule matches, false otherwise.  
 \*/  
function evaluateMatchRules(condition, value) {  
  // No condition to evaluate, rule passes.  
  if (condition == '') {  
    return true;  
  }  
  var rules = [matchesBelow, matchesAbove, matchesRange];  
  
  for (var i = 0; i < rules.length; i++) {  
    if (rules[i](condition, value)) {  
      return true;  
    }  
  }  
  return false;  
}  
  
/\*\*  
 \* Evaluates whether a value is below a threshold value.  
 \* @param {string} condition The condition to be checked. (e.g. below 50).  
 \* @param {number} value The value to be checked.  
 \* @return {boolean} True if the value is less than what is specified in  
 \* condition, false otherwise.  
 \*/  
function matchesBelow(condition, value) {  
  conditionParts = condition.split(' ');  
  
  if (conditionParts.length != 2) {  
    return false;  
  }  
  
  if (conditionParts[0] != 'below') {  
    return false;  
  }  
  
  if (value < conditionParts[1]) {  
    return true;  
  }  
  return false;  
}  
  
/\*\*  
 \* Evaluates whether a value is above a threshold value.  
 \* @param {string} condition The condition to be checked. (e.g. above 50).  
 \* @param {number} value The value to be checked.  
 \* @return {boolean} True if the value is greater than what is specified in  
 \*     condition, false otherwise.  
 \*/  
function matchesAbove(condition, value) {  
  conditionParts = condition.split(' ');  
  
  if (conditionParts.length != 2) {  
    return false;  
  }  
  
  if (conditionParts[0] != 'above') {  
    return false;  
  }  
  
  if (value > conditionParts[1]) {  
    return true;  
  }  
  return false;  
}  
  
/\*\*  
 \* Evaluates whether a value is within a range of values.  
 \* @param {string} condition The condition to be checked (e.g. 5 to 18).  
 \* @param {number} value The value to be checked.  
 \* @return {boolean} True if the value is in the desired range, false otherwise.  
 \*/  
function matchesRange(condition, value) {  
  conditionParts = condition.replace('\w+', ' ').split(' ');  
  
  if (conditionParts.length != 3) {  
    return false;  
  }  
  
  if (conditionParts[1] != 'to') {  
    return false;  
  }  
  
  if (conditionParts[0] <= value && value <= conditionParts[2]) {  
    return true;  
  }  
  return false;  
}  
  
/\*\*  
 \* Retrieves the weather for a given location, using the Weather Underground  
 \* API.  
 \* @param {string} location The location to get the weather for.  
 \* @return {Object.<string, string>} The weather attributes and values, as  
 \*     defined in the API.  
 \*/  
function getWeather(location) {  
  if (location in WEATHER\_LOOKUP\_CACHE) {  
    Logger.log('Cache hit...');  
    return WEATHER\_LOOKUP\_CACHE[location];  
  }  
  
  var url = Utilities.formatString(  
      'http://api.openweathermap.org/data/2.5/weather?APPID=%s&q=%s',  
      encodeURIComponent(OPEN\_WEATHER\_MAP\_API\_KEY),  
      encodeURIComponent(location));  
  var response = UrlFetchApp.fetch(url);  
  if (response.getResponseCode() != 200) {  
    throw Utilities.formatString(  
        'Error returned by API: %s, Location searched: %s.',  
        response.getContentText(), location);  
  }  
  
  var result = JSON.parse(response.getContentText());  
  
  // OpenWeatherMap's way of returning errors.  
  if (result.cod != 200) {  
    throw Utilities.formatString(  
        'Error returned by API: %s,  Location searched: %s.',  
        response.getContentText(), location);  
  }  
  
  WEATHER\_LOOKUP\_CACHE[location] = result;  
  return result;  
}  
  
/\*\*  
 \* Adjusts the bidModifier for a list of geo codes for a campaign.  
 \* @param {string} campaignName The name of the campaign.  
 \* @param {Array} geocodes The list of geo codes for which bids should be  
 \*     adjusted.  
 \* @param {number} bidModifier The bid modifier to use.  
 \*/  
function adjustBids(campaignName, geocodes, bidModifier) {  
  // Get the campaign.  
  var campaignIterator = AdWordsApp.campaigns().withCondition(  
      Utilities.formatString('CampaignName = "%s"', campaignName)).get();  
  while (campaignIterator.hasNext()) {  
    var campaign = campaignIterator.next();  
  
    // Get the targeted locations.  
    var locations = campaign.targeting().targetedLocations().get();  
    while (locations.hasNext()) {  
      var location = locations.next();  
      var currentBidModifier = location.getBidModifier().toFixed(2);  
  
      // Apply the bid modifier only if the campaign has a custom targeting  
      // for this geo location.  
      if (geocodes.indexOf(location.getId()) != -1 &&  
          currentBidModifier != bidModifier) {  
        Logger.log('Setting bidModifier = %s for campaign name = %s, ' +  
            'geoCode = %s. Old bid modifier is %s.', bidModifier, campaignName,  
            location.getId(), currentBidModifier);  
        location.setBidModifier(bidModifier);  
      }  
    }  
  }  
}