**Owen Galvin, CSCI E-88 Final Project**

**Summary**

**Goals:** Demonstrate ability to stream data into Kafka and consume in a column store database, Vertica. Vertica has native machine learning capabilities that can be very fast for large datasets compared to the alternatives. Visualization of data pulled from Vertica is performed via various libraries (Bokeh, Seaborn, Matplotlib) in a Jupyter notebook.

|  |
| --- |
|  |

**Data:** The immediate data source is a series of anonymized credit card transactions available on Kaggle website. As cited on the dataset's overview page:

*The dataset has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (Université Libre de Bruxelles) on big data mining and fraud detection.*

**Pipeline:** Credit card transaction data is sent to Kafka, simulating a stream of data, by loading into a pandas dataframe and sending out JSON-formatted batches via kafka-python. For Vertica the Kafka Connect connector is configured via a set of SQL statements, which in the end set up a Scheduler daemon that constantly monitors Kafka for a specific topic or topics. Once the data has been ingested the machine learning functions of Vertica are used on a training subset to create a couple of logistic regression models with different baseline configurations. From there a Jupyter notebook is used for both exploratory/demonstrative purposes and also to evaluate the generated models.

**Results:** Since the primary purpose of the project was to demonstrate the data pipeline I would say everything was successful. A reach goal would have been to have some live monitoring of the Kafka streaming, perhaps even with predictive modeling, on the data that was held out from initial ingestion and I didn't get at all close to that time-wise. Also I'm not sure how the logistic models faired and didn't evaluate beyond some confusion matrixes but considered those operations to be ancillary to project goals anyway.

**youtube:** <https://youtu.be/5ReZBnM_x4w>

**Environment**

My primary "computing" environment is a Linux (CentOS 7) virtual machine running on Windows 10 laptop via VirtualBox. As the VM can be rather laggy in terms of UI responsiveness (has been assigned two of my cores + 16GB of RAM, actual computation is fine) I've set up port forwarding so that Vertica can be accessed via DBeaver and Jupyter notebook on my host OS.

Prerequisites etc.

* Kafka, Zookeeper
  + installed via previous assignment
* Vertica
  + v 8.1.1 had been installed on my VM some months ago
  + links to documentation are presented later on
* Python
  + pandas, numpy, kafka-python, bokeh, matplotlib, seaborn
  + jupyter notebook, Anaconda environment

In terms of actual timeline almost all of the visualization parts weren't developed until the end. Those were created in a Jupyter notebook and attempt to intersperse this document with the relevant parts, adding text and or screenshots in an offset box.

Here are the bits that directly relate to imports etc.

JUPYTER

|  |
| --- |
| **Overview**  The general idea of the visualization part was to play around with the Bokeh library in Python, defaulting to matplotlib if necessary, looks like I also briefly visited seaborn. Turns out to not have been the best time for an introduction to Bokeh as that project is undergoing significant restructuring as it remains a low-level API and significant functionality is moved into the more high-level HoloViews API, specifically the bokeh.charts has been deprecated but my code still uses that module.  **import** pandas **as** pd  **import** numpy **as** np  **import** matplotlib  **import** matplotlib.pyplot **as** plt  **%**matplotlib inline  **from** sklearn.metrics **import** confusion\_matrix  **import** seaborn **as** sn  **import** bokeh  **from** bokeh.io **import** output\_notebook  In [4]:  Bokeh is undergoing some significant changes (see BokehDeprecationWarning below) and many of the chart examples on the web won't work with the current version. I started with that latest version, then regressed to 0.12.0, and then back up to 0.12.6 in order to get some additional features.  In [5]:  print('pandas version:', pd.\_\_version\_\_)  print('bokeh version: ', bokeh.\_\_version\_\_)  pandas version: 0.21.0  bokeh version: 0.12.6  In [17]  **from** bokeh.charts **import** Bar, Histogram, Donut, output\_file, show, color  **from** bokeh.palettes **import** Spectral6, brewer  **from** bokeh.io **import** output\_notebook  **from** bokeh.models.mappers **import** LinearColorMapper  **from** bokeh.models **import** ColorBar  **from** bokeh.layouts **import** row  *# need this to output bokeh charts via Jupyter*  output\_notebook()  ​  *# for final project display suppress all warning messages*  **import** warnings  warnings.simplefilter('ignore') |
| C:\Anaconda3\envs\conda\_v35\_e88\_misc\lib\site-packages\bokeh\util\deprecation.py:34: BokehDeprecationWarning:  The bokeh.charts API has moved to a separate 'bkcharts' package.  This compatibility shim will remain until Bokeh 1.0 is released.  After that, if you want to use this API you will have to install  the bkcharts package explicitly.  warn(message) |

**The Data**

The source data for this project comes from Kaggle.com and is described as " Anonymized credit card transactions labeled as fraudulent or genuine". It is available for download at <https://www.kaggle.com/dalpozz/creditcardfraud/downloads/creditcard.csv> but a free account would be required in order to access.

There are 284,807 transactions in the .csv, with 30 variables, plus the fraud indicator. 28 of these variables are numerical values that represent the result of PCA (Primary Component Analysis) transformation, a statistical technique used in the creation of predictive models. The remaining two fields are Time (first record = 0, subsequent observations contain the number of seconds from this first transaction) and Amount, which I'm assuming is in euros.

**Note:** the dataset is highly unbalanced, with only 492 fraudulent transactions out of the 284k total.

**creditcard.csv**, header and first row of data, first 100 rows will be included in project submission

|  |
| --- |
| Time","V1","V2","V3","V4","V5","V6","V7","V8","V9","V10","V11","V12","V13","V14","V15","V16","V17","V18","V19","V20","V21","V22","V23","V24","V25","V26","V27","V28","Amount","Class"  0,-1.3598071336738,-0.0727811733098497,2.53634673796914,1.37815522427443,-0.338320769942518,0.462387777762292,0.239598554061257,0.0986979012610507,0.363786969611213,0.0907941719789316,-0.551599533260813,-0.617800855762348,-0.991389847235408,-0.311169353699879,1.46817697209427,-0.470400525259478,0.207971241929242,0.0257905801985591,0.403992960255733,0.251412098239705,-0.018306777944153,0.277837575558899,-0.110473910188767,0.0669280749146731,0.128539358273528,-0.189114843888824,0.133558376740387,-0.0210530534538215,149.62,"0" |

**Stream to Kafka**

The first step in terms of Kafka is the running of zookeeper, followed by kafka, as was discussed in Assignment 4.

**zookeeper-server-start.sh $KAFKA\_HOME/config/zookeeper.properties**

**kafka-server-start.sh $KAFKA\_HOME/config/server.properties**

With those up and running in separate terminal windows I create a new topic with 2 partitions named "credit\_fraud"

**[cloudera@localhost proj]$ kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 2 --topic credit\_fraud**

WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('\_') could collide. To avoid issues it is best to use either, but not both.

Created topic "credit\_fraud".

Below is the python script I'll be using to simulate streaming the data into Kafak

**proj\_producer.py**

|  |
| --- |
| **import** argparse **import** time **import** datetime  **import** pandas **as** pd **from** kafka **import** KafkaConsumer, KafkaProducer    **def** send\_events(topic\_name, batch\_count, start\_at, stop\_at, is\_debug):   print(**'loading csv ...'**)  data = pd.read\_csv(**'/mnt/share/e88/creditcard.csv'**, sep=**','**)  producer = KafkaProducer(bootstrap\_servers=**'localhost:9092'**)   **for** i **in** range(start\_at, stop\_at, batch\_count):  batch = data[i:i + batch\_count]  json\_events = batch.to\_json(orient=**'records'**)  **if** is\_debug:  print(json\_events)  producer.send(topic\_name, str.encode(json\_events))  print(i, i+batch\_count, len(batch))  print(**'-'** \* 44)   *#time.sleep(1)*   producer.flush()   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'Send json events to Kafka'**)  parser.add\_argument(**'-t'**, **'--topic'**, type=str)  parser.add\_argument(**'-b'**, **'--batch-count'**, type=int, default=5)  parser.add\_argument(**'-s'**, **'--start-at'**, type=int, default=0)  parser.add\_argument(**'-e'**, **'--end-at'**, type=int, default=30)  parser.add\_argument(**'-d'**, **'--debug'**, type=bool, default=**False**)  args = parser.parse\_args()    print(**'BEGIN TEST, topic: {}, batch-count: {}, start-at: {}, end-at: {}, debug: {}'**.format(  args.topic, args.batch\_count,args.start\_at, args.end\_at, args.debug))  print(**'sending first batch at {}'**.format(datetime.datetime.now()))  send\_events(args.topic, args.batch\_count, args.start\_at, args.end\_at, args.debug) |

A series of command line arguments indicate which topic to use, along with the batch size and the begin/end row indices of the transactions to be sent to Kafka via the kafka-python library. The csv is loaded into a pandas dataframe, which makes conversion to json (this is what Vertica will expect) very easy. An array of events with size = the batch-count is sliced off of the dataframe with each loop, exiting once enough batches have been submitted to cover the start/end values.

There is a sleep for one second in there for aesthetic/simulation purposes, will remove when loading large amounts of data. The syntax of the start/end resembles that of a Python list in that "--end-at" would me more accurately called "stop-by", i.e. using "...-s 0 -e 10..." would start at row with index 0 and continue to row index 9, not actually including the row at index 10.

A quick test before going any further.

Begin with a terminal running the kafka consumer with a topic = "credit\_fraud" (bottom window)

Next run the python script with 2 batches of 10 transactions each.

**(v34\_e88) [cloudera@localhost proj]$ python proj\_producer.py -t credit\_fraud -b 10 -s 0 -e 20 -d True**



**Vertica Analytic Database**

Vertica is a column-oriented database system owned by HP and is perhaps most directly comparable to Amazon's Redshift. It has a free Community Edition, limited to 1 terabyte of data and that is the version (Vertica Analytic Database v8.1.1-0) that I had installed onto my Linux VM several months ago. The install process in theory is relatively straightforward and covered at [Installing Vertica](https://my.vertica.com/docs/8.1.x/HTML/index.htm#Authoring/InstallationGuide/InstallingVertica/DownloadAndInstallTheVerticaInstallPackage.htm%3FTocPath%3DInstalling%2520Vertica%7CInstalling%2520Vertica%7CInstalling%2520Using%2520the%2520Command%2520Line%7C_____1) though my notes indicate I ran into some problems related to installing it on a resource-limited virtual machine.

We use Vertica at my workplace but I'd never used it in terms of Kafka before (or used any of its Machine Learning features).

**Kafka Connect for Vertica**

The main page for Kafka Connect at <https://www.confluent.io/product/connectors/> lists both a sink a source under the Certified Connectors listing for Vertica. The hyperlinks there are rather outdated and the documentation relating to Kafka and Vertica for the 8.1 version I have is located at [Integrating with Apache Kafka](https://my.vertica.com/docs/8.1.x/HTML/index.htm#Authoring/KafkaIntegrationGuide/KafkaIntegrationGuide.htm%3FTocPath%3DIntegrating%2520with%2520Apache%2520Kafka%7C_____0). Overall I found the doucmentation to be rather confusing and lacking in a basic end-to-end example, often using different configuration values in the documentation snippets. At times I thought perhaps they didn't want people to actually use the Connector yet - the feature is present so it could be advertised, but it was difficult enough to set up that they would only need to support a limited customer base until any problems were ironed out. (I believe the Kafka link was first available with version 7.2 of Vertica, the latest 9.0 version came out several weeks ago.)

Configuration for Kafka is generally performed via the command line vkconfig script, beginning with creating a parent "scheduler" schema for the kafka data. (The below steps are performed as sudo of the Vertica superuser = dbadmin.) The separate utility functions, designated by the first argument to the script, have many options but my setup involves a minimum of non-default settings.

1. create the scheduler that continously monitors Kafka for data to load into Vertica

/opt/vertica/packages/kafka/bin/vkconfig scheduler --add --config-schema kafka\_proj --operator dbadmin

A query on the db reveals the 11 tables that Vertica created as a result

**SELECT** table\_name **FROM** tables **WHERE** table\_schema = 'kafka\_proj';

stream\_scheduler

stream\_clusters

stream\_sources

stream\_targets

stream\_load\_specs

stream\_microbatches

stream\_microbatch\_source\_map

stream\_microbatch\_history

stream\_events

stream\_lock

stream\_scheduler\_history

1. create the cluster object - this is the item that allows Vertica to connect to multiple clusters.

/opt/vertica/packages/kafka/bin/vkconfig cluster --create --cluster kafka1 --config-schema kafka\_proj --hosts localhost:9092

In my case of course this points to the only (local) cluster running on localhost.

**NOTE: config-schema** argument, with value matching the scheduler created in step 1, is used here and in most calls to vkconfig. If this argument is not included Vertica will apparently default to using the stream\_config schema, the default scheduler present for streaming. The script may complete without error but the configuration will have taken place on this default scheduler instead of the "active" one, which caused me some trouble while figuring things out.

1. create the source, with a value (and partition count) matching that of the Kafka topic

/opt/vertica/packages/kafka/bin/vkconfig source --create --config-schema kafka\_proj --source credit\_fraud --cluster kafka1 --partitions 2

Older documentation uses a "topic" utility along with an argument named topic. Presumably they are trying to generalize the concept in order to support multiple streaming systems in the future.

1. now over to some SQL in Vertica, need to create the table that will receive Kafka data

**CREATE** **TABLE** proj.credit\_fraud (

**Time** **FLOAT**,

V1 **FLOAT**,

V2 **FLOAT**,

V3 **FLOAT**,

V4 **FLOAT**,

V5 **FLOAT**,

V6 **FLOAT**,

V7 **FLOAT**,

V8 **FLOAT**,

V9 **FLOAT**,

V10 **FLOAT**,

V11 **FLOAT**,

V12 **FLOAT**,

V13 **FLOAT**,

V14 **FLOAT**,

V15 **FLOAT**,

V16 **FLOAT**,

V17 **FLOAT**,

V18 **FLOAT**,

V19 **FLOAT**,

V20 **FLOAT**,

V21 **FLOAT**,

V22 **FLOAT**,

V23 **FLOAT**,

V24 **FLOAT**,

V25 **FLOAT**,

V26 **FLOAT**,

V27 **FLOAT**,

V28 **FLOAT**,

Amount **FLOAT**,

Class **INTEGER**

)

Schema matches that of source csv. I tried a shortcut by running something like below after loading the csv into a Jupyter notebook for analysis, pandas has been aliased as pd:

ddl = pd.io.sql.get\_schema(data, **'credit\_fraud'**)  
print(ddl.replace(**'REAL'**, **'FLOAT'**).replace(**'"'**,**""**))

I had initially experimented with what Vertica calls a flex-table, where the raw event from Kafka would have been stored in a schema-less table, basically as text. Then the schema can be imposed on the data via a view. I had trouble setting up the view properly though and reverted to this more straightforward workflow.

1. create the target, i.e. the Vertica table created in above step

/opt/vertica/packages/kafka/bin/vkconfig target --create --config-schema kafka\_proj --target-schema proj --target-table credit\_fraud

1. create a load specification that matches the format of incoming stream

/opt/vertica/packages/kafka/bin/vkconfig load-spec --create --config-schema kafka\_proj --load-spec ls\_credit\_fraud --parser KafkaJSONParser

My python script earlier formats the data into JSON, so a matching parser is selected here. In v8.1 of Vertica the other parser options are: KafkaParser, KafkaJSONParser, or KafkaAvroParser

1. define a microbatch, which is used to execute the actual COPY statements that pull data from the stream into a Vertica table

/opt/vertica/packages/kafka/bin/vkconfig microbatch --create --config-schema kafka\_proj --microbatch mb\_credit\_fraud --target-schema proj --target-table credit\_fraud --rejection-schema proj --rejection-table credit\_fraud\_rej --load-spec ls\_credit\_fraud --add-source credit\_fraud --add-source-cluster kafka1

This is basically a combination of values seen in most of the preceding steps, tying everything together. New items include a rejection schema & table, which will store any events that fail to load into Vertica, along with reason for failure - very useful for troubleshooting.

1. start the scheduler, which tells Vertica to begin monitoring for Kafka events (the scheduler stays resident in a given terminal window until stopped)

/opt/vertica/packages/kafka/bin/vkconfig launch --conf /home/dbadmin/configFile.properties --instance-name launch\_credit\_fraud --config-schema kafka\_proj

The configFile.properties files contains information on further CLI options that include username & password. Below is from the documentation:

**Important:** Micro Focus International plc does not recommend specifying a password on the command line. Instead, put the password in a properties file.

1. not part of the the initial configuration but important to note that all schedulers can be safely shut down by running command like below in separate shell

/opt/vertica/packages/kafka/bin/vkconfig shutdown

Finally, one other utility option I played around with relates to offset control. Again there are many more settings available for fine tuning and troubleshooting,

/opt/vertica/packages/kafka/bin/vkconfig microbatch --update --microbatch mb\_credit\_fraud --offset -2,-2 --partition 0,1 --config-schema kafka\_proj

The above would reset the offset on both partitions (number 0 and 1) back to the very beginning (value = -2 for both). As described in the documentation.

|  |  |
| --- | --- |
|  | The offset of the message in the source where the microbatch starts its load. If you use this parameter, you must supply an offset value for each partition in the source or each partition you list in the --partition option.  You can use this option to skip some messages in the source or reload previously read messages. |

**Streaming**

Now that everything has been configured and the scheduler is up and running per step 8) above events can be sent to Kafka, and from there consumed by Vertica. I ran my producer script two times, the first with two batches of 10 rows, covering rows 0 to 20, and then a second time for the 10 rows after that. Below is the truncated debug output of the second run.

**(v34\_e88) [cloudera@localhost proj]$ python proj\_producer.py -t credit\_fraud -b 10 -s 20 -e 30 -d True**

BEGIN TEST, topic: credit\_fraud, batch-count: 10, start-at: 20, end-at: 30, debug: True

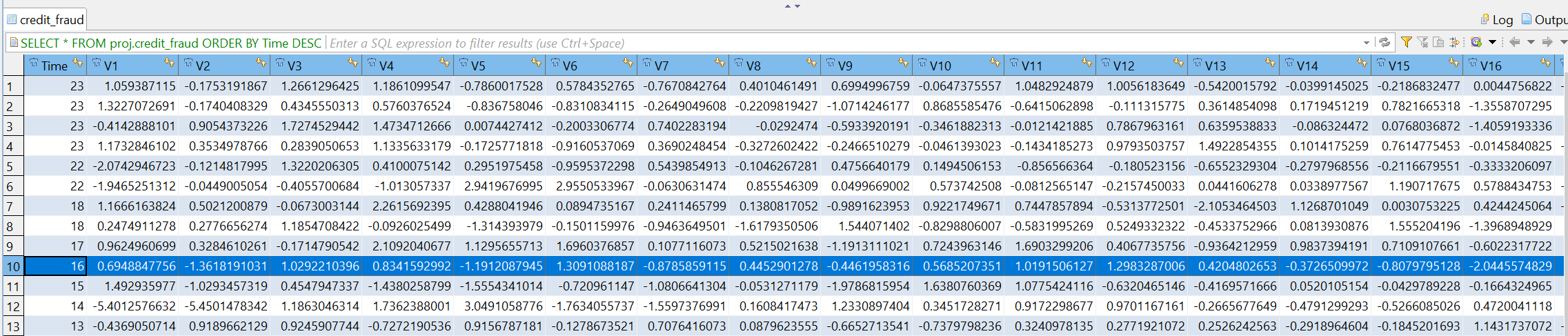
sending first batch at 2017-12-06 21:33:49.291609

loading csv ...

[{"Time":16.0,"V1":0.6948847756,"V2":-1.3618191031,"V3":1.0292210396,"V4":0.8341592992,"V5":-1.1912087945,"V6":1.3091088187,"V7":-0.8785859115,"V8":0.4452901278,"V9":-0.4461958316,"V10":0.5685207351,"V11":1.0191506127,"V12":1.2983287006,"V13":0.4204802653,"V14":-0.3726509972,"V15":-0.8079795128,"V16":-2.0445574829,"V17":0.515663469,"V18":0.6258472984,"V19":-1.3004081688,"V20":-0.1383339404,"V21":-0.2955829316,"V22":-0.5719550068,"V23":-0.0508807005,"V24":-0.304214501,"V25":0.0720010061,"V26":-0.4222344304,"V27":0.0865533981,"V28":0.0634986493,"Amount":231.71,"Class":0},{"Time":17.0,"V1":0.9624960699,"V2":0.3284610261,"V3":-0.1714790542,"V4":2.1092040677,"V5":1.1295655713,"V6":1.6960376857,"V7":0.1077116073,"V8":0.5215021638,"V9":-1.1913111021,"V10":0.7243963146,"V11":1.6903299206,"V12":0.4067735756,"V13":-0.9364212959,"V14":0.9837394191,"V15":0.7109107661,"V16":-0.6022317722,"V17":0.4024843756,"V18":-1.7371620345,"V19":-2.0276123218,"V20":-0.2693209665,"V21":0.1439974234,"V22":0.4024916614,"V23":-0.0485082212,"V24":-1.3718662945,"V25":0.3908138854,"V26":0.1999636575,"V27":0.0163706433,"V28":-0.0146053277,"Amount":34.09,"Class":0},{"Time":18.0,"V1":1.1666163824,"V2":0.5021200879,"V3":-0.0673003144,"V4":2.2....

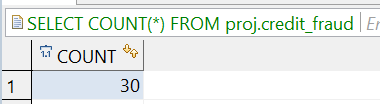
From DBeaver, the GUI app I'm using to communicate with Vertica, I look at he target table that had been manually created earlier in order to receive the events.

**SELECT** \* **FROM** proj.credit\_fraud **ORDER** **BY** **Time** **DESC**



The events are listed in reverse order per my query and we can see that the values of the highlighted row (10th from top, i.e. the first row sent from that last producer run) match that of the debug output.

**SELECT** **COUNT**(\*) **FROM** proj.credit\_fraud

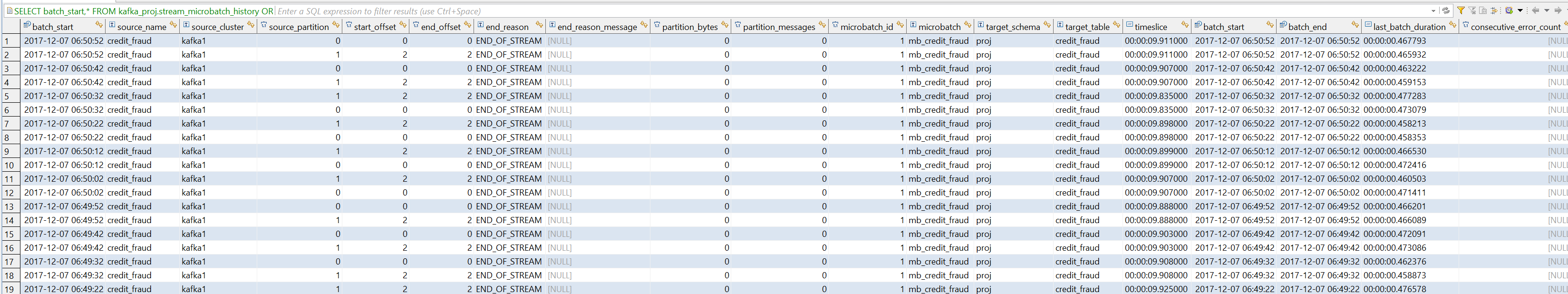


The creation of the Scheduler object had led to a schema of that same name being created in Vertica, the resulting tables had been listed earlier in the doc. One of the more informative metadata tables was stream\_microbatch\_history, which is described here: [This table contains a history of every microbatch executed within this scheduler configuration](https://my.vertica.com/docs/8.1.x/HTML/index.htm#Authoring/KafkaIntegrationGuide/KafkaTables/stream_microbatch_history.htm) and copied down below since I think it gives a good feel for the monitoring capabilites within Vertica.

| **Column** | **Data Type** | **Description** |
| --- | --- | --- |
| source\_name | VARCHAR | The name of the source. |
| source\_cluster | VARCHAR | The name of the source cluster. To query sources, refer to [stream\_sources](https://my.vertica.com/docs/8.1.x/HTML/Content/Authoring/KafkaIntegrationGuide/KafkaTables/stream_sources.htm). |
| source\_partition | INTEGER | The number of the data streaming partition. |
| start\_offset | INTEGER | The starting offset of the microbatch. |
| end\_offset | INTEGER | The ending offset of the microbatch. |
| end\_reason | VARCHAR | An explanation for why the batch ended. The following are valid end reasons:   * DEADLINE - The batch ran out of time * END\_OFFSET - The load reached the ending offset specified in the KafkaSource * END\_OF\_STREAM - There are no messages available to the scheduler or the eof\_timeout has been reached * NETWORK\_ERROR - The scheduler could not connect to Kafka * RESET\_OFFSET - The start offset was changed using the --update and --offset parameters to the KafkaSource. This state does not occur during normal scheduler operations. * SOURCE\_ERROR - The specified Kafka topic does not exist * UNKNOWN - The batch ended for an unknown reason |
| end\_reason\_message | VARCHAR | If the end reason is a network or source issue, this column contains a brief description of the issue. |
| partition\_bytes | INTEGER | The number of bytes transferred from a source partition to a Vertica target table. |
| partition\_messages | INTEGER | The number of messages transferred from a source partition to a Vertica target table. |
| microbatch\_id | INTEGER | The Vertica transaction id for the batch session. |
| microbatch | VARCHAR | The name of the microbatch. |
| target\_schema | VARCHAR | The name of the target schema. |
| target\_table | VARCHAR | The name of the target table. |
| timeslice | INTERVAL | The amount of time spent in the KafkaSource operator. |
| batch\_start | TIMESTAMP | The time the batch executed. |
| batch\_end | TIMESTAMP | The time the batch completed. |
| last\_batch\_duration | INTERVAL | The length of time required to run the complete COPY statement. |
| consecutive\_error\_count | INTEGER | The number of times a microbatch has encountered an error on an attempt to load. This value increases over multiple attempts. |
| transaction\_id | INTEGER | The identifier for the transaction within the session. |
| frame\_start | TIMESTAMP | The time the frame started. A frame can contain multiple microbatches. |
| frame\_end | TIMESTAMP | The time the frame completed. |

A peek into that same table reveals info related to the 30 first events

**SELECT** batch\_start,\* **FROM** kafka\_proj.stream\_microbatch\_history **ORDER** **BY** batch\_start **DESC**



After confirming there is no sleep pause in between batch runs I go ahead and load in balance of first 250k rows from the .csv. This leaves off about 34k rows, which may later be loaded if I have time to come up with an interesting scenaro, 250k is also a round number I can use for sanity checks as I learn more about machine learning in Vertica.

**python proj\_producer.py -t credit\_fraud -b 10 -s 30 -e 250000**

...

249900 249910 10

--------------------------------------------

249910 249920 10

--------------------------------------------

249920 249930 10

--------------------------------------------

249930 249940 10

--------------------------------------------

249940 249950 10

--------------------------------------------

249950 249960 10

--------------------------------------------

249960 249970 10

--------------------------------------------

249970 249980 10

--------------------------------------------

249980 249990 10

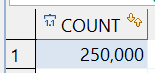
--------------------------------------------

249990 250000 10

--------------------------------------------

Confirm they all made it into Vertica, which of course means there isn't any reason to check that rejection table for problems

**SELECT** **COUNT**(\*) **FROM** proj.credit\_fraud



**Learning**

The thrust of this final project is the data workflow, not any data manipulation within one of the tiers ,so I'll run through the model creation etc. within Vertica in a relatively cursory manner.

The first step is to split the data into train and test sets, so that a model can be created based on one set of data (training) and then evaluated on a disparate set of similar data (testing), going with a 70/30 split.

**CREATE** **TABLE** proj.credit\_fraud\_split **AS**

**SELECT** \*,

**CASE** **WHEN** RANDOM() < 0.3 **THEN** 'test' **ELSE** 'train' **END** **AS** part

**FROM** proj.credit\_fraud

**CREATE** **TABLE** proj.credit\_fraud\_train **AS**

**SELECT** \* **FROM** proj.credit\_fraud\_split **WHERE** part = 'train';

**CREATE** **TABLE** proj.credit\_fraud\_test **AS**

**SELECT** \* **FROM** proj.credit\_fraud\_split **WHERE** part = 'test'

**SELECT** **COUNT**(\*) **FROM** proj.credit\_fraud\_train

**UNION** **ALL**

**SELECT** **COUNT**(\*) **FROM** proj.credit\_fraud\_test

> 174951

> 75049

Now to balance the training dataset - if the full data were to be analzyed as-is we would likely wind up with a model that always predicted non-fraud. For the ingested dataset this would be well over 99% accuracy but wouldn't be helpful in terms of the problem set.

|  |  |
| --- | --- |
| **SELECT** Class, **COUNT**(\*) **as** cnt **from** proj.credit\_fraud **GROUP** **BY** Class | **SELECT** 1 - 458/(249542+458) |

Luckily\* Vertica 8.1 includes a BALANCE function designed exactly handle this scenario, which is not uncommon in machine learning.

-- 'name\_of\_new\_balanced\_view', 'name\_of\_source\_table', 'column\_to\_balance', sampling-method

-- "weighted\_sampling" is only sampling option in Vertica 8.0, more in this versio n= 8.1 hybrid\_sampling over\_sampling under\_sampling alias=weighted\_sampling

-- optional: sampling\_ratio=valueThe ratio of the sample size to population size. Default value: 0.5

**SELECT** BALANCE('proj.credit\_fraud\_train\_balanced\_view', 'proj.credit\_fraud\_train', 'Class', 'under\_sampling ' **USING** PARAMETERS sampling\_ratio=0.85);

Move data from view into static table.

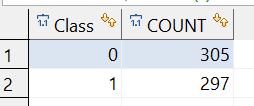
**CREATE** **TABLE** proj.credit\_fraud\_train\_balanced\_ **AS**

**SELECT** \* **FROM** proj.credit\_fraud\_train\_balanced\_view

**SELECT** Class, **COUNT**(\*)

**FROM** proj.credit\_fraud\_train\_balanced

**GROUP** **BY** Class

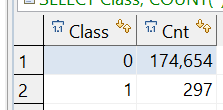


Per-Class counts for the dataset before the balancing.

**SELECT** Class, **COUNT**(\*) **as** Cnt

**FROM** proj.credit\_fraud\_train

**GROUP** **BY** Class



I told the BALANCE to create a 50/50 dataset and that is what it did by undersampling - kept all 297 fraudulent transactions from the train table and then randomly selected about the same number from the non-fraudulent rows of that same table

|  |
| --- |
| \* Unluckily the BALANCE function doesn't seem to actually work properly in my Vertica instance. Perhaps there is a bug or perhaps somehow related to running only a single node cluster but I found the created view, which itself was defined with a RANDOM calculation, only worked on the first few SELECT statements, after that it started returning only records of one class. Either way, I persisted the results of an early view query into a regular table and didn't need to worry about it afterwards. |

JUPYTER

|  |
| --- |
| Pull some of the baseline data in from Vertica via vertica\_python library.  **import** ssl  **import** vertica\_python  ​​  ssl\_context = ssl.SSLContext(ssl.PROTOCOL\_SSLv23)  ssl\_context.verify\_mode = ssl.CERT\_NONE  ssl\_context.check\_hostname = **False**  ​  *# port forwarding of 5433 on VM to host OS*  conn\_info = {  'host': 'localhost',  'port': 5433,  'user': 'dbadmin',  'password': vertica\_pwd,  'database': 'vert'  *# ,'ssl': ssl\_context,*  }  ​  **with** vertica\_python.connect(**\*\***conn\_info) **as** conn:  data = pd.read\_sql\_query('SELECT \* from proj.credit\_fraud', conn)  train = pd.read\_sql\_query('SELECT \* from proj.credit\_fraud\_train\_balanced', conn)  ​  train.head()   Exploration Beginning with the full dataset, do some exploration. First viz is very simple without any formatting. Notable: 1) the raw count values are very imbalanced such that it appears as if all the transactions are in one of the first two low-euro-amount bins 2) usually takes around 8 secons to generate the chart  **%%**time  hist = Histogram(data, values='Amount', title="Amount distribution", bins=20, plot\_width=700)  show(hist)    Next is a much fancier histogram that shares a similar base purpose:   1. The transformation of raw data into histogram is done on the db side    * much quicker than above, usually 10x at least, and that is with relatively small dataset    * some queries may be easier to write in SQL, either due to inherent features or because of user knowledge 2. Euro amounts are displayed on a logarithmic scale, obvious now that most transactions are clustered in the low Amount range but that there are a few distibuted across the full $0 - $20k range 3. Bin color is based on the relative occurrence of fraudulent transactions within a bin    * the first bin likely has the highest raw number of fraudulent charges but the coloring reveals that as a percentage most fraudulent charges are occurring in the (Avg) $1400 bin    * in fact there appear to be no fraudulent transactions greater than somewhere around $3k    * below query confirms, max fraudulent charge = $2,125.87   SELECT MAX(Amount) FROM proj.credit\_fraud WHERE Class = 1  **%%**time  HIST\_SQL = """  -- https://stackoverflow.com/questions/44994183/vertica-generate-table-with-numbers-select-prime-numbers  WITH numbers AS (  SELECT ROW\_NUMBER() OVER() AS num FROM (  SELECT 1 FROM (  SELECT date(0) + INTERVAL '1 second' AS se UNION ALL  SELECT date(0) + INTERVAL '1000 seconds' AS se ) a  TIMESERIES tm AS '1 second' OVER(ORDER BY se)  ) b  )  --SELECT num FROM seq ;  ,bins AS (  SELECT bin, COUNT(\*) AS BinSize, AVG(Amount)::int as BinAvgAmount, AVG(Class) \* 1000 as FraudRatio  FROM (  SELECT Amount, Class, WIDTH\_BUCKET(Amount, 0, (SELECT MAX(Amount) FROM proj.credit\_fraud), 19) as bin  FROM proj.credit\_fraud  ) A  GROUP BY bin  ORDER BY bin  )  SELECT n.num as bin, COALESCE(BinSize, 0) as BinSize,  CASE WHEN BinSize = 1 THEN 0.4 ELSE COALESCE(LN(BinSize), 0) END as LogBinSize,  'Bin ' || RIGHT('0' || n.num::char(2), 2) ||  CASE WHEN COALESCE(BinAvgAmount, 0) = 0 THEN ' NA'  ELSE ' $' || COALESCE(BinAvgAmount, 0)::char(10)  END as BinAvgAmount,  COALESCE(WIDTH\_BUCKET(FraudRatio, 0, 5, 6), 1) as FraudRatio  FROM numbers n  LEFT JOIN bins b ON b.bin = n.num  WHERE n.num <= 20  ORDER BY n.num  """  ​  **with** vertica\_python.connect(**\*\***conn\_info) **as** conn:  hist\_data = pd.read\_sql\_query(HIST\_SQL, conn)  ​  print(hist\_data[['bin','BinSize','LogBinSize','BinAvgAmount']].head(20))  ​  color\_mapper = LinearColorMapper(palette=brewer['RdYlGn'][6], low=0, high=10)  color\_bar = ColorBar(color\_mapper=color\_mapper, location=(**-**85, 0), height=300,  title='Relative Fraud')  color\_bar.title\_text\_align = 'center'  ​  bar = Bar(hist\_data, values='LogBinSize', label='BinAvgAmount', agg='mean', legend=**None**,  xlabel='Avg $ Amount in bin', ylabel='Log Amount',  color=color(columns=['FraudRatio'], palette=brewer['RdYlGn'][4]),  title="(Log) Amount Distribution", plot\_width=800)  bar.add\_layout(color\_bar, 'right')  show(bar)  ​  bin BinSize LogBinSize BinAvgAmount  0 1 247472 12.419053 Bin 01 $73  1 2 1944 7.572503 Bin 02 $1403  2 3 335 5.814131 Bin 03 $2511  3 4 149 5.003946 Bin 04 $3572  4 5 50 3.912023 Bin 05 $4554  5 6 21 3.044522 Bin 06 $5603  6 7 10 2.302585 Bin 07 $6719  7 8 10 2.302585 Bin 08 $7696  8 9 3 1.098612 Bin 09 $8646  9 10 1 0.400000 Bin 10 $10000  10 11 0 0.000000 Bin 11 NA  11 12 2 0.693147 Bin 12 $11844  12 13 1 0.400000 Bin 13 $12911  13 14 0 0.000000 Bin 14 NA  14 15 0 0.000000 Bin 15 NA  15 16 0 0.000000 Bin 16 NA  16 17 0 0.000000 Bin 17 NA  17 18 0 0.000000 Bin 18 NA  18 19 1 0.400000 Bin 19 $18910  19 20 1 0.400000 Bin 20 $19657 |

|  |
| --- |
| Balancing Output some info on the training datasets, both the original and the balanced version  In [8]:  COUNT\_SQL="""  SELECT Class, COUNT(\*) As Cnt FROM proj.{table\_name} GROUP BY Class  """  sql\_count\_train\_raw = COUNT\_SQL.format(table\_name='credit\_fraud\_train')  sql\_count\_train\_balanced = COUNT\_SQL.format(table\_name='credit\_fraud\_train\_balanced')  ​  **with** vertica\_python.connect(**\*\***conn\_info) **as** conn:  train\_class\_count\_raw = pd.read\_sql\_query(sql\_count\_train\_raw, conn)  train\_class\_count\_balanced = pd.read\_sql\_query(sql\_count\_train\_balanced, conn)  ​  print('preliminary cut for training dataset')  print(train\_class\_count\_raw)  print()  print('balanced dataset used for modeling')  print(train\_class\_count\_balanced)  preliminary cut for training dataset  Class Cnt  0 0 174654  1 1 297  balanced dataset used for modeling  Class Cnt  0 0 305  1 1 297  Below are matplotlib pie charts showing the distribution of fraudulent to non-fraudulent transactions in the training datasets   1. on left is the full 175k dataset, fraud makes up a tiny fraction 2. on the far right is the balanced dataset, much more closer in proportion but many fewer transactions overall 3. nestled in the middle is the balanced pie chart, but in an overall area approximating the the size of the balanced training dataset vs. the unbalanced   In [9]:  fig = plt.figure(figsize=(15, 5))  colors=['green', 'cyan']  ​  ax1 = fig.add\_subplot(131)  ax1.pie(train\_class\_count\_raw['Cnt'],  labels=['Class=0 / No Fraud: {}'.format(train\_class\_count\_raw['Cnt'][0])  ,'Class=1 / Fraud: {}'.format(train\_class\_count\_raw['Cnt'][1])],  colors=colors, autopct='%1.1f%%', shadow=**True**, startangle=90)  ​  ax2 = fig.add\_subplot(133)  ax2.pie(train\_class\_count\_balanced['Cnt'],  labels=['Class=0 / No Fraud: {}'.format(train\_class\_count\_balanced['Cnt'][0])  ,'Class=1 / Fraud: {}'.format(train\_class\_count\_balanced['Cnt'][1])],  colors=colors, autopct='%1.1f%%', shadow=**True**, startangle=90)  ​  ax3 = fig.add\_subplot(132)  ax3.pie(train\_class\_count\_balanced['Cnt'], startangle=90, colors=colors, radius=0.0583095)  ​  plt.show()    Next is a very simple bokeh version, noting that:   1. the Donut chart, as with other bokeh.chart objects, has been moved into the lightly-maintained bkcharts project 2. there isn't much configuration that can be done on Donut from what I could tell, perhaps HoloViews has a whole new implementation 3. there are much more fine grained ways of creating a pie chart, this one was very easy though   In [10]:  pie\_chart\_raw = Donut(train\_class\_count\_raw, values='Cnt', label='Class', text\_font\_size='10pt')  pie\_chart\_balanced = Donut(train\_class\_count\_balanced, values='Cnt', label='Class', text\_font\_size='10pt')  ​  show(row(pie\_chart\_raw, pie\_chart\_balanced) ) |

Now to create a couple of logistic regression models, potenitally many different settings, with many possible values (for something like lambda), but for a demo two models will be fine, differing only by the optimizer used.

--SYNTAX: LOGISTIC\_REG ( 'model\_name', 'input\_relation', 'response\_column', 'predictor\_columns'

-- [ USING PARAMETERS[exclude\_columns='col1, col2, ... coln',] [optimizer='value',] [epsilon=value,] [max\_iterations=value] [regularization= 'value',] [lambda= value] ])

-- more info: https://my.vertica.com/docs/8.1.x/HTML/index.htm#Authoring/SQLReferenceManual/Functions/MachineLearning/LOGISTIC\_REG.htm

-- optimizer: Newwton (default) or BFGS (aka Broyden–Fletcher–Goldfarb–Shanno)

**SELECT** LOGISTIC\_REG('proj.log\_model\_1', 'proj.credit\_fraud\_train\_balanced', 'Class', '\*' **USING** PARAMETERS exclude\_columns='Class, part, balanced');

-- Return value: Finished in 30 iterations

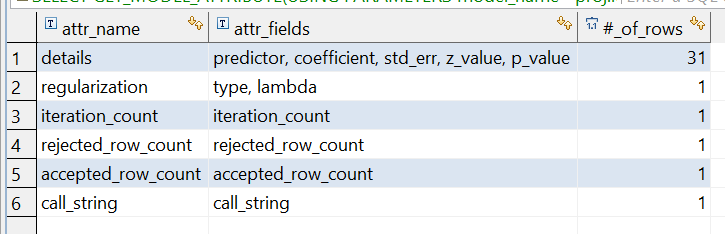
**SELECT** LOGISTIC\_REG('proj.log\_model\_2', 'proj.credit\_fraud\_train\_balanced', 'Class', '\*' **USING** PARAMETERS exclude\_columns='Class, part, balanced', optimizer='BFGS');

-- Return value: Finished in 100 iterations

I'll run through a few functions that are available in terms of examining the models.

-- what attributes are available

**SELECT** GET\_MODEL\_ATTRIBUTE(**USING** PARAMETERS model\_name='proj.log\_model\_1');



Look at the details on both models

|  |  |
| --- | --- |
| **SELECT** GET\_MODEL\_ATTRIBUTE(**USING** PARAMETERS model\_name='proj.log\_model\_1', attr\_name='details'); | **SELECT** GET\_MODEL\_ATTRIBUTE(**USING** PARAMETERS model\_name='proj.log\_model\_2', attr\_name='details'); |
| ... |  |

Display further details on the first model

**SELECT** SUMMARIZE\_MODEL('proj.log\_model\_1');

|  |
| --- |
| coeff names: {Intercept, **time**, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, amount}  coefficients: {-172.3778175, -2.086667797e-05, 12.76260379, 289.5952098, -160.3655934, 119.653972, -30.86916761, -133.9062358, -488.8800566, 85.48566285, -163.8224983, -378.6999394, 307.1205622, -551.1488583, -10.60461337, -596.4910336, -19.13012634, -526.1307259, -933.2954969, -355.7083381, 141.8774418, -69.83384159, 32.3545604, 54.81844449, 156.4655047, -16.86324219, 76.1759913, 17.96583951, 74.80119577, 198.1964867, 1.841609879}  std\_err: {9284, 1.121e-05, 2016, 1.368e+04, 5291, 4316, 478.5, 6220, 2.142e+04, 3635, 6536, 1.503e+04, 1.276e+04, 2.291e+04, 604.3, 2.501e+04, 893.4, 2.206e+04, 3.872e+04, 1.48e+04, 6140, 4160, 993.9, 2720, 8220, 783.3, 3745, 943.3, 2938, 9974, 95}  z\_value: {-0.01857, -1.862, 0.006332, 0.02117, -0.03031, 0.02772, -0.06451, -0.02153, -0.02283, 0.02352, -0.02506, -0.0252, 0.02407, -0.02405, -0.01755, -0.02385, -0.02141, -0.02385, -0.02411, -0.02404, 0.02311, -0.01679, 0.03255, 0.02015, 0.01904, -0.02153, 0.02034, 0.01905, 0.02546, 0.01987, 0.01939}  p\_value: {0.9852, 0.06266, 0.9949, 0.9831, 0.9758, 0.9779, 0.9486, 0.9828, 0.9818, 0.9812, 0.98, 0.9799, 0.9808, 0.9808, 0.986, 0.981, 0.9829, 0.981, 0.9808, 0.9808, 0.9816, 0.9866, 0.974, 0.9839, 0.9848, 0.9828, 0.9838, 0.9848, 0.9797, 0.9841, 0.9845}  Regularization **method**: **none**, lambda: 1  **Number** **of** iterations: 30, **Number** **of** skipped samples: 0, **Number** **of** processed samples: 602  **Call**:  logistic\_reg('proj.log\_model\_1', 'proj.credit\_fraud\_train\_balanced', '"class"', '\*'  **USING** PARAMETERS exclude\_columns='Class, part, balanced', optimizer='newton', epsilon=1e-06, max\_iterations=100, regularization='none', lambda=1) |

Finaly, display information related to the how successful the models are when evaluated on the test dataset. After the earlier split proj.credit\_fraud\_test wound up with 74,888 non-fraudulent transactions + 161 that were fraud.

The top cte represents the base query, modify the SELECT under the cte to get at underlying outputs one at a time. A default cutoff value of 0.5 is used even though in a more real world scenario the value used for evaluation would likely be higher or lower. Do we discount the penalty of false positives, preferring to label a transaction as fraudulent so that it can be further evaluated via some other model? Or if this is some kind of realtime evaluation where it might be better to only block transactions that we are very certain are fraudulent. In both scenarios the Amount value would probably be factored into decision making, i.e the "cost" of missing a fraudulent high euro amount vs. low euro amount.

**WITH** cte **AS** (

**SELECT** 'model\_1' **as** model, Amount, Class, PREDICT\_LOGISTIC\_REG(**Time**,V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15,V16,V17,V18,V19,V20,V21,V22,V23,V24,V25,V26,V27,V28,Amount

**USING** PARAMETERS model\_name='proj.log\_model\_1', **type**='response', cutoff=0.5)::**INT** **as** PredClass

**FROM** proj.credit\_fraud\_test

**UNION** **ALL**

**SELECT** 'model\_2', Amount, Class, PREDICT\_LOGISTIC\_REG(**Time**,V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15,V16,V17,V18,V19,V20,V21,V22,V23,V24,V25,V26,V27,V28,Amount

**USING** PARAMETERS model\_name='proj.log\_model\_2', **type**='response', cutoff=0.5)::**INT** **as** PredClass

**FROM** proj.credit\_fraud\_test

)

**SELECT**

-- "" one or the other on below, else: "ERROR: Cannot specify more than one user-defined transform function in the SELECT list"

CONFUSION\_MATRIX(Class, PredClass **USING** PARAMETERS num\_classes=2) **OVER**()

-- ERROR\_RATE(Class, PredClass USING PARAMETERS num\_classes=2) OVER()

**FROM** cte

**WHERE** model = 'model\_1'

|  |  |  |
| --- | --- | --- |
|  | model\_1 | model\_2 |
| CONFUSION\_MATRIX output |  |  |
| ERROR\_RATE  (last row appears to be weighted average error rate) |  |  |

JUPYTER

|  |
| --- |
| Model analysis Displaying confustion matrix graphics for each of the various predictive models (and cutoffs in this case) can help in judging model accuracy, by displaying true/false positive and true/false negative counts. The PREDICT\_LOGISTIC\_REG function in Vertica is used to actually make predictions, but using different cutoff values as discussed in the Word document, where different cutoffs may be desirables depending on business reasoning. The CONFUSION\_MATRIX function of Vertica is also called, more as a demonstration than anything - the actual details involved in the calculation are straightforward.  In [11]:  PRED\_SQL="""  SELECT Class, PREDICT\_LOGISTIC\_REG(Time,V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15,V16,V17,V18,V19,V20,V21,V22,V23,V24,V25,V26,V27,V28,Amount  USING PARAMETERS model\_name='proj.log\_model\_1', cutoff={cutoff})::INT as PredClass  FROM proj.{table\_name}  """  sql\_test\_pred01 = PRED\_SQL.format(cutoff=0.1, table\_name='credit\_fraud\_test')  sql\_test\_pred25 = PRED\_SQL.format(cutoff=0.25, table\_name='credit\_fraud\_test')  sql\_test\_pred50 = PRED\_SQL.format(cutoff=0.5, table\_name='credit\_fraud\_test')  ​  CM\_SQL="""  SELECT CONFUSION\_MATRIX(Class, PredClass USING PARAMETERS num\_classes=2) OVER() FROM ({pred\_sql}) A  """  ​  **with** vertica\_python.connect(**\*\***conn\_info) **as** conn:  cm\_test\_data\_01 = pd.read\_sql\_query(sql\_test\_pred01, conn)  cm\_test\_data\_25 = pd.read\_sql\_query(sql\_test\_pred25, conn)  cm\_test\_data\_50 = pd.read\_sql\_query(sql\_test\_pred50, conn)  cm\_01 = pd.read\_sql\_query(CM\_SQL.format(pred\_sql=sql\_test\_pred01), conn)    print(cm\_test\_data\_01.head())  print()  print(cm\_01)  Class PredClass  0 0 0  1 0 0  2 0 0  3 0 0  4 0 1  class 0 1 comment  0 0 65082 9806  1 1 6 155 Of 75049 rows, 75049 were used and 0 were ignored  With the various dataframes populated the next step will be to create a series of confusion matrixes for each of the cutoffs. A confusion matrix on a highly imbalanced dataset is unlikely to be very useful...  In [12]:  *# function definition used in earlier class I took, CS109A*  **def** plot\_confusion\_matrix(confusion, ax, title='Confusion matrix',cmap=plt.cm.gray\_r):    tot = float(confusion[0,0] **+** confusion[0,1] **+** confusion[1,0] **+** confusion[1,1])    conf = np.empty([2,2])    conf[0,0] = float(confusion[0,0]) **/** tot  conf[0,1] = float(confusion[0,1]) **/** tot  conf[1,0] = float(confusion[1,0]) **/** tot  conf[1,1] = float(confusion[1,1]) **/** tot  ​  cax = ax.matshow(conf, cmap=cmap)  ax.set\_title(title)  *#fig.colorbar(cax)*  tick\_marks = np.arange(len(conf[0]))  plt.xticks(tick\_marks)  plt.yticks(tick\_marks )    ax.set\_ylabel('Actual')  ax.set\_xlabel('Predicted')  plt.text(0,2, confusion, size='large')    **return** ax  ​  ​  fig = plt.figure(figsize=[15,8])  ax = fig.add\_subplot(131)  cf\_01 = confusion\_matrix(cm\_test\_data\_01['Class'],cm\_test\_data\_01['PredClass'])  plot\_confusion\_matrix(cf\_01, ax, 'Log Reg Confusion cutoff=0.1\n')  print(cf\_01)  ​  ax2 = fig.add\_subplot(132)  cf\_25 = confusion\_matrix(cm\_test\_data\_25['Class'],cm\_test\_data\_25['PredClass'])  plot\_confusion\_matrix(cf\_25, ax2, 'Log Reg Confusion cutoff=0.25\n')  print(cf\_25)  ​  ax3 = fig.add\_subplot(133)  cf\_50 = confusion\_matrix(cm\_test\_data\_50['Class'],cm\_test\_data\_50['PredClass'])  plot\_confusion\_matrix(cf\_50, ax3, 'Log Reg Confusion cutoff=0.50\n')  print(cf\_50)  ​  plt.show()  [[65082 9806]  [ 6 155]]  [[69339 5549]  [ 8 153]]  [[71738 3150]  [ 9 152]]    The imbalanced nature of the full test dataset indeed makes the output less than useful. Nonetheless, go ahead and try a different library instead of matplotlib. Bokeh didn't seem to have anything out-of-the-box (perhaps because I was only searching for "confusion matrix" instead of heatmap) but Seaborndid. Code is very simple so worth a shot. Create a single one based on the datframe returned by Vertica's CONFUSION\_MATRIX.  In [31]:  sn.set(font\_scale=1.5)  sn.heatmap(cm\_01[['0','1']], annot=**True**, annot\_kws={'size': 14}, cmap='Greys', fmt='g')  plt.show()    This one is probably better because the numbers are embedded withing the graphic and the color scale comes by default (and it was basically one line of code). In terms of generating matrixes for actual evaluation though it would be better to use a balanced test dataset as the source.  In [32]:  sql\_test\_balanced\_pred01 = PRED\_SQL.format(cutoff=0.1, table\_name='credit\_fraud\_test\_balanced')  sql\_test\_balanced\_pred25 = PRED\_SQL.format(cutoff=0.25, table\_name='credit\_fraud\_test\_balanced')  sql\_test\_balanced\_pred50 = PRED\_SQL.format(cutoff=0.5, table\_name='credit\_fraud\_test\_balanced')  ​  **with** vertica\_python.connect(**\*\***conn\_info) **as** conn:  cm\_test\_balanced\_data\_01 = pd.read\_sql\_query(sql\_test\_balanced\_pred01, conn)  cm\_test\_balanced\_data\_25 = pd.read\_sql\_query(sql\_test\_balanced\_pred25, conn)  cm\_test\_balanced\_data\_50 = pd.read\_sql\_query(sql\_test\_balanced\_pred50, conn)    In [33]  fig = plt.figure(figsize=[15,8])  ax = fig.add\_subplot(131)  cf\_01 = confusion\_matrix(cm\_test\_balanced\_data\_01['Class'],cm\_test\_balanced\_data\_01['PredClass'])  plot\_confusion\_matrix(cf\_01, ax, 'Log Reg test\_balanced cutoff=0.1\n')  print(cf\_01)  ​  ax2 = fig.add\_subplot(132)  cf\_25 = confusion\_matrix(cm\_test\_balanced\_data\_25['Class'],cm\_test\_balanced\_data\_25['PredClass'])  plot\_confusion\_matrix(cf\_25, ax2, 'Log Reg test\_balanced cutoff=0.25\n')  print(cf\_25)  ​  ax3 = fig.add\_subplot(133)  cf\_50 = confusion\_matrix(cm\_test\_balanced\_data\_50['Class'],cm\_test\_balanced\_data\_50['PredClass'])  plot\_confusion\_matrix(cf\_50, ax3, 'Log Reg test\_balanced cutoff=0.50\n')  print(cf\_50)  ​  plt.show()  [[138 16]  [ 6 155]]  [[146 8]  [ 8 153]]  [[150 4]  [ 9 152]]    Two of the balanced datasets, now in seaborn, with some color added in this time.  In [80]:  fig, (ax1,ax2) = plt.subplots(ncols=2, figsize=[12,4])  sn.heatmap(cf\_01, annot=**True**, annot\_kws={'size': 14}, cmap='Reds', fmt='g', ax=ax1)  sn.heatmap(cf\_25, annot=**True**, annot\_kws={'size': 14}, cmap='Reds', fmt='g', ax=ax2)  ​  print(cf\_01)  print(cf\_25)  plt.show()  [[138 16]  [ 6 155]]  [[146 8]  [ 8 153]] |

**Conclusion**

Vertica's Kafka Connect mostly consists of running a script with a multiplicity of options and there isn't really a programmatic way of interacting on the Vertica side. The configuration, and the storage of said configuration within Vertica tables, seemed a bit unwieldly but I don't have much of a basis for comparison. In a real world trial I would want multiple topics, with different settings, running over a relatively lengthy period of time. Vertica itself as a storage tier component seems very competent, though I did have trouble when my VM crashed, corrupting the Vertica database I had been using. My experience at work has shown it to be extremely fast at certain types of queries, even with huge amounts of data. The machine learning part was kind of a side project and I'm not sure how flexible it is in an ad-hoc manner. The performance may be great but to a degree you are limited to what functions Vertica provides + SQL. The combination of something like pandas for further manipulation of the outputs seemed promising. Finally, I wanted to look at either Bokeh or Plotly for visualization with Python and went with the former mostly because I preferred its name. I didn't get to go as deep into Bokeh as I had hoped but I've used matplotlib a bit but that package seems rather dated to what else is out there nowadays. If I were looking at something for a real world implementation though I would likely avoid Bokeh for the next few months, until the separation/integration between it and the higher level HoloViews becomes more clear (and there are more examples available on the web).