Credit Card Fraud Detection

**Introduction**

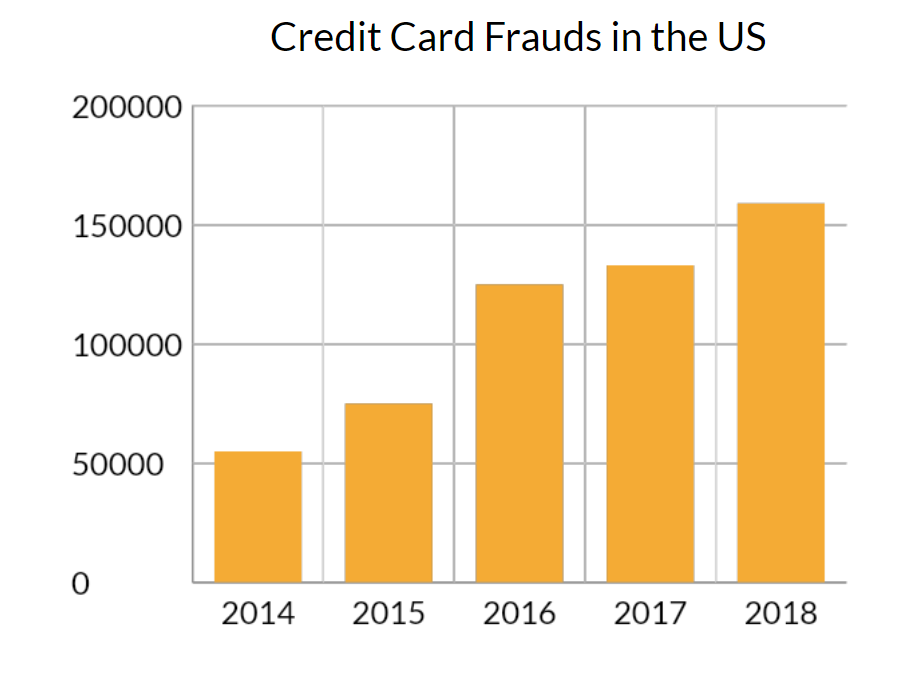
Welcome to the Capstone Project. This project revolves around the most widely used tools used in the Big Data engineering world.

**Problem Statement**

Before moving onto the problem statement, you need to understand what credit card fraud is and why there is a need to detect these fraudulent transactions. Then, you will take a look at the problem statement.

**Note**: In the video below, there was a problem with the SME's presentation software due to which the bullet points are not working correctly.

Credit card fraud is defined as a form of identity theft in which an individual uses someone else’s credit card information to make purchases or to withdraw funds from the account. The incidence of such fraudulent transactions has skyrocketed as the world has moved towards a digital era. The following statistics will help you understand the gravity of the situation.



Credit Card Frauds

With the rising number of fraud cases, the company’s major focus is to provide its customers with a delightful experience while ensuring that security is not compromised.

As a big data engineer, you need to architect and build a solution to cater to the following requirements:

1. **Fraud detection solution**: This is a feature to detect fraudulent transactions, wherein once a cardmember swipes their card for payment, the transaction is classified as fraudulent or authentic based on a set of predefined rules. If fraud is detected, then the transaction must be declined. Please note that incorrectly classifying a transaction as fraudulent will incur huge losses to the company and also provoke negative consumer sentiment.
2. **Customer information**: The relevant information about the customers needs to be continuously updated on a platform from where the customer support team can retrieve relevant information in real-time to resolve customer complaints and queries.

Additional Reading

[Credit Card Fraud](https://legaldictionary.net/credit-card-fraud/) - An article on Credit Card Fraud and its elements.

**Data**

The following tables containing data will be taken into consideration to solve this problem:

* **card\_member**(The cardholder’s data is stored in a central AWS RDS.)
  + card\_id: This refers to the card number.
  + member\_id: This is the 15-digit member ID of the cardholder.
  + member\_joining\_dt: This is the date and time of joining of new member.
  + card\_purchase\_dt: This is the date on which the card was purchased.
  + country: This is the country in which the card was purchased.
  + city: This is the city in which the card was purchased.
* **card\_transactions**(All incoming transactions (fraud/genuine) swiped at point of sale (POS) terminals are stored in this table.)
  + card\_id: This refers to the card number.
  + member\_id: This is the 15-digit member ID of the cardholder.
  + amount: This is the amount that is swiped with respect to the card\_id.
  + postcode: This is the ZIP code at which this card was swiped (marking the location of an event).
  + pos\_id: This is the merchant’s POS terminal ID, using which the card was swiped.
  + transaction\_dt: This is the date and time of the transaction.
  + status: This indicates whether the transaction was approved or not, with a genuine/fraud value.
* **member\_score**(The member credit score data is stored in a central AWS RDS.)
  + member\_id: This is the 15-digit member ID of the cardholder.
  + score: This is the score assigned to a member defining their credit history, generated by upstream systems.

Data related to **card\_member**and **member\_score**is stored in a central AWS RDS. You will be given the **card\_transactions**data, which has already been classified, in the form of a CSV file, which you can load in your NoSQL database.

The other type of data is the real-time streaming data that is generated by the POS systems in a JSON format. The streaming data looks like this:

* Transactional payload (data) attributes sent by POS terminals’ gateway API on to the Kafka topic:
  + card\_id: This is the card number.
  + member\_id: This is the 15-digit member ID of the cardholder.
  + amount: This is the amount that is swiped with respect to the card\_id.
  + pos\_id: This is the merchant’s POS terminal ID, using which the card was swiped.
  + postcode: This is the ZIP code at which this card was swiped (marking the location of an event).
  + transaction\_dt: This is the date and time of the transaction.

Here is an example of a JSON payload structure that gets produced.

{

"card\_id":**348702330256514**,

"member\_id": **000037495066290**,

"amount": **9084849**,

"pos\_id": **614677375609919**,

"postcode": **33946**,

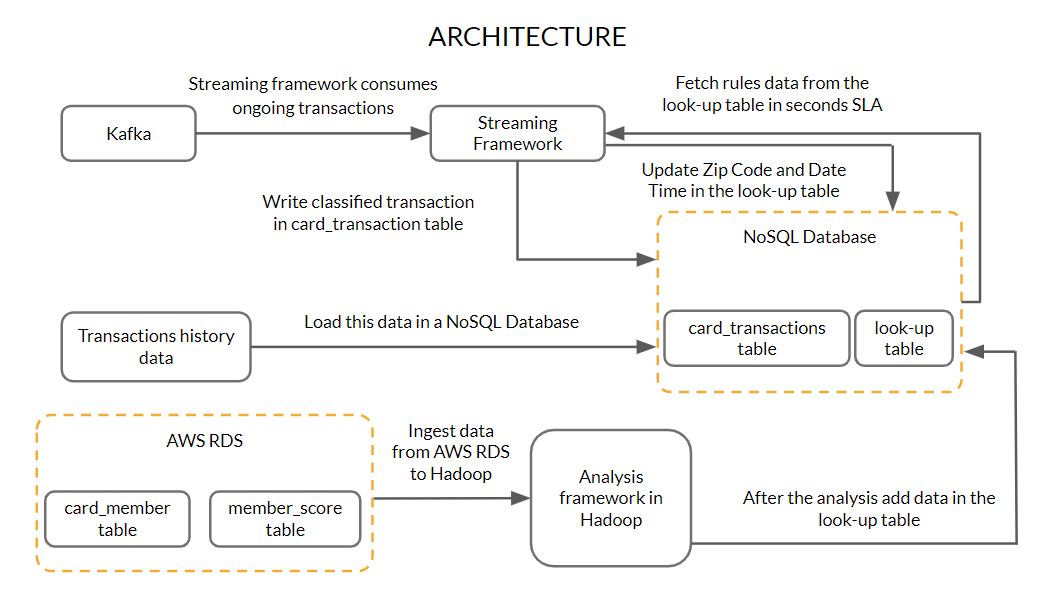
"transaction\_dt": "11-02-2018 00:00:00"

}

**Architecture and Approach**

Having understood the various kinds of data involved in this project, it is time for you to understand the architecture of building a solution to this problem statement. Let’s hear about it from Sagan in the next video.

Let us look at the architecture in little more detail.



Architecture

The details of the member and the credit score associated with members are hosted on a central AWS RDS server. The historical transaction data will be provided as a CSV file. You need to use appropriate ingestion methods available to bring the card\_member and member\_score data from the AWS RDS into a Hadoop platform. You also need to load the historical card transactions into a NoSQL database. This data is then processed to fill data in the look-up table.

Now, the data from the several POS systems will flow inside the architecture through a queuing system such as Kafka. The POS data from Kafka will be consumed by the streaming data processing framework to identify the authenticity of the transactions.

You should note that one of the Service-Level Agreement (SLAs) of the company is to complete the transaction within a few seconds. Once the POS data from Kafka is entered into the stream processing layer, it is then assessed based on some parameters defined by the rules. The values for these parameters are fetched from the look-up table. The transaction is allowed to complete only when the results are positive for these rules. If the result for any rule is negative, then the transaction should be classified as fraud.

Once the transaction is classified as genuine, then, corresponding to the card ID in the look-up table, the postcode and the transaction date of the current transaction need to be updated as per the last transaction. These fields should only be updated if the transaction gets classified as genuine.

The card\_transactions table also needs to be updated with all the details along with the classification of the transactions.

The lookup table will contain the following details:

* Card id
* Upper control limit (UCL)
* Postcode of the last transaction
* Transaction date of the last transaction
* The credit score of the member

Now, let’s understand the various parameters defined by the rules required to determine the authenticity of transactions. Here are the three parameters that we will use to detect whether a transaction is fraudulent or not.

1. **Upper control limit (UCL):** Every card user has an upper limit on the amount per transaction, which is different from the maximum transaction limit on each card. This parameter is an indicator of the transaction pattern associated with a particular customer. This upper bound, also known as the upper control limit (UCL), can be used as a parameter to authenticate a transaction. Suppose you have a past record of making transactions with an average amount of $20,000, and one day, the system observes a transaction of $200,000 through your card. This can be a possible case of fraud. In such cases, the cardholder receives a call from the credit card company executives to validate the transaction. UCL is derived using the following formula:

UCL = Moving average + 3 × (Standard deviation)

This formula is used to derive the UCL value for each card\_id. The moving average and the standard deviation for each card\_id are calculated based on the last 10 amounts credited that were classified as genuine.

**Note:** If the total number of transactions for a particular card\_id is less than 10, then calculate the parameters based on the total number of records available for that card\_id.

2. **Credit score of each member**: This is a straightforward rule, where you have a member\_score table in which member IDs and their respective scores are available. These scores are updated by a third-party service. If the score is less than 200, that member’s transaction is rejected, as they could be a defaulter. This rule simply defines the financial reputation of each customer.

3. **ZIP code distance**: The whole purpose of this rule is to keep a check on the distance between the card owner's current and last transaction location with respect to time. If the distance between the current transaction and the last transaction location with respect to time is greater than a particular threshold, then this raises suspicion on the authenticity of the transaction. Suppose at time t = t0 minutes, a transaction is recorded in Mumbai, and at time t

= (t0 + 10) minutes, a transaction from the same card\_id is recorded in New York. A flight flies with a cruising speed of about 900 km/hr, which means that someone travelling by Airbus can travel a kilometre in four seconds. This (a transaction in Mumbai followed by one in New York after 10 minutes) can be a possible case of fraud. Such cases happen often, when someone acquires your credit card details and makes transactions online. In such cases, the cardholder receives a call from the credit card company executive to validate the transaction.

Now that you have a fair understanding of these parameters, let’s discuss the approach to calculating them.

Let’s start with the **upper control limit (UCL)**. The historical transactional data is stored in the **card\_transactions**table, which was defined earlier. The UCL value needs to be calculated for each card\_id for the last 10 transactions. One approach could be to trigger the computation of this parameter for a card\_id every time a transaction occurs. However, considering the few seconds SLA, this might not be a good practice, as batch jobs are always associated with huge time delays.

Another approach could be to have a lookup table that stores the UCL values based on the moving average and standard deviation of the last 10 transactions for each card\_id. Whenever a transaction occurs, the record corresponding to the card\_id can be easily fetched from this lookup table, rather than calculating the UCL value at the time of the transaction.

The second parameter is based on the credit score of the member. If this score is less than 200, then the transaction needs to be declined, as the member could turn out to be a defaulter.

The third parameter is based on the ZIP code analysis. Store the ‘postcode’ and ‘transaction\_dt’ parameters pertaining to the last transaction for each card\_id in the look-up table. Whenever a new transaction occurs, retrieve the ‘postcode’ and ‘transaction\_dt’ attributes from the look-up table and compare these with the current ‘postcode’ and ‘transaction\_dt’ data. Use the API to calculate the speed at which the user moved from the origin. If it is more than the imaginable speed, this can be a possible case of fraud. In such cases, the cardholder receives a call from the credit card company executive to validate the transaction.

After initiating the real-time process following each member’s transaction, update the current received transaction’s ‘postcode’ and ‘transaction\_dt’ as the last ZIP code and time in the lookup table stored in the NoSQL database if and only if the transaction is approved (satisfying all three rules).

Once a transaction is evaluated based on the aforementioned three parameters, the transaction, along with the status (i.e., genuine or fraud) of the transaction, is stored in the card\_transactions table in the database.

Once you start the Kafka consumer in the streaming framework, each transaction of different members will be iterated and checked for these rules without any lag.

Now in the next segment, you will learn about the different tasks that you will be working on in this capstone project.

**Tasks**

Before we discuss the guidelines and tasks that you need to perform, make sure that you are configuring the EMR cluster properly.

You will need to make sure that you have **Hadoop, Sqoop, Hive, HBase and Spark** installed on your cluster with **Hue**as an optional service. Also as an added step, make sure that in the **Hardware configuration step** for the EMR cluster generation, scroll down to the **EBS Root Volume configuration** and type the **Root device EBS volume size** as **20 GB.**

In the next video, Sagan will discuss the tasks that you need to complete as part of this project.

**Note**: In the video below, there was a problem with the SME's presentation software due to which the bullet points are not working correctly.

As part of the project, broadly, you are required to perform the following tasks:

* **Task 1**: Load the transactions history data (card\_transactions.csv) in a NoSQL database.
* **Task 2**: Ingest the relevant data from AWS RDS to Hadoop.
* **Task 3**: Create a look-up table with columns specified earlier in the problem statement.
* **Task 4**: After creating the table, you need to load the relevant data in the lookup table.
* **Task 5**: Create a streaming data processing framework that ingests real-time POS transaction data from Kafka. The transaction data is then validated based on the three rules’ parameters (stored in the NoSQL database) discussed previously.
* **Task 6**: Update the transactions data along with the status (fraud/genuine) in the card\_transactions table.
* **Task 7**: Store the ‘postcode’ and ‘transaction\_dt’ of the current transaction in the look-up table in the NoSQL database if the transaction was classified as genuine.

**Note:**At the end of **three weeks**you are required to make submissions for the **first four tasks**. Then at the end of **six weeks**, you are required to make submissions for the **last three tasks**. The relevant documents related to these submissions are present in the '**Submissions**' segment in the next session.

**Note:**The details to get these data from RDS and Kafka is present in the '**Resources**' segment.

Validation

1. When you load the data of the past card transactions in the NoSQL database the count of the data should be **53,292**. This will be same as the number of records present in the **card\_transactions.csv** file.
2. When you run the sqoop jobs to import the data from AWS RDS it would retrieve 999 records.
3. When you classify all the incoming transactions as fraud or genuine and then update this in the card\_transactions table, the final count of that table should be more than **59,000**.

**Resources**

The RDS connection string and credentials are as follows:

* RDS **Connection String** -
* jdbc:mysql://upgradawsrds1.cyaielc9bmnf.us-east-1.rds.amazonaws.com/cred\_financials\_data
* **Username** - upgraduser
* **Password** - upgraduser
* **Database -**cred\_financials\_data
* **Table Name** - card\_member and member\_score

To connect to Kafka use the following details:

* **Bootstrap-server:**18.211.252.152
* **Port Number:**9092
* **Topic**: transactions-topic-verified

**Note:**Do not create your own Kafka cluster. Also, **do not produce anything on the Kafka topic provided on your own. This can have a serious effect on your grading.**

Use the following python code and the zipCodePosId.csv file to calculate the distance between two postcodes:

**Distance Utility**

**Download**

**uszipsv.csv**

**Download**

Dump of card\_transactions table that needs to be loaded in a NoSQL database:

**card\_transactions**

**Download**

To connect to HBase tables and access the tables in HBase use the following python code:

**HBase**

**Download**

* Make sure that the Thrift server is running.
* Provide the Public IP of your EC2 Instance in place of the IP address "54.86.61.43" as mentioned in the file shared.

In the next session, you will get to know about the Grading Rubrics for this capstone project and the Submission guidelines of the project.

**Note**: Please refer to the Kafka Integration segment in the Industry demo session in the Spark Streaming module to see how to read and write to Kafka with the help of Spark Streaming.

**Submission**

**Note**: For the mid submission, you need to complete and submit the files corresponding to the first 6 points and upload the .zip file for the same. The sample submission docs are attached below for your reference.

**Submissions required**

Upload a zip file containing the following:

1. ​​​​A PDF document (**LoadNoSQL.pdf**) containing the commands to load the transactions history data (card\_transactions.csv) in a NoSQL database.
2. A PDF document (**SqoopDataIngestion.pdf**) containing the code used for ingesting data from the RDS server.
3. A PDF document (**CreateNoSQL.pdf**) to create a look-up table with columns specified earlier in the problem statement.
4. Script to calculate the moving average and standard deviation of the last 10 transactions for each card\_id for the data present in Hadoop and NoSQL database. If the total number of transactions for a particular card\_id is less than 10, then calculate the parameters based on the total number of records available for that card\_id. The script should be able to extract and feed the other relevant data (‘postcode’, ‘transaction\_dt’, ‘score’, etc.) for the look-up table along with card\_id and UCL. (**PreAnalysis.pdf**)  
   **Note:** In this pdf provide all the commands to load the lookup table with the relevant data.
5. Screenshots of the execution of the scripts written. The scripts should, after loading the data and creating the look-up table, take the data from the NoSQL database and AWS RDS and perform the relevant analyses as per the rules and should feed the data in the look-up table (**ScriptsExecution.pdf**).
6. Explanation of the solution to the batch layer problem in detail should be provided properly in a document. (**LogicMid.pdf**)  
   **Note: All the sample files have been provided to you as a word document at the end of this segment in a zipped file.**
7. The structure of your directory should be as described ahead. A directory named "**python**" should be created. It should contain a directory named "**src**". The "**src**" directory should have two directories named "**db**" and "**rules**".  The "**src**" directory should have a python file named "**driver.py**" which should be the calling the other files and should be the entry point of your code. The "**rules**" directory should contain a "**rules.py**" file where you write the functions to check for the three rules mention earlier. The "**db**" directory should have the "**geo\_map.py**" and "**dao.py**" shared earlier. The **driver.py**file should contain the code to read the messages from Kafka and call necessary functions from the python files present in the "**rules**"and "**db**"directory to classify the incoming transaction as fraud or genuine.  
   **Note: It is not necessary to follow the above guidelines. Whatever the code you write, you need to properly comment it and also document the exact steps to run the code provided by you.**  
   **Note: It is very important to provide a document to explain the code that you have written for the streaming layer. Also, you need to properly document the steps to execute the same code.**
8. Explanation of the solution to the streaming layer problem in detail should be provided properly in a document. (**LogicFinal.pdf**)  
   **Note: This is a must. You need to clearly explain the code and the steps to run the same should be properly documented.**

**Please make sure that you are not changing any of the file names that have been provided above in parentheses. The code that you are submitting should run at our end without any modifications in the code.**

The following zip file contains all the sample documents.

You must go through these guidelines-

1. Make sure you have not made any changes to the original dataset provided to you. Your code should work on the dataset given to you as part of the problem statement. You are not allowed to make modifications in data set using excel and then use it in your Python code. Entire data processing must be done in, as mentioned only. During grading we will be running your code on the dataset provided by us, in case your code gives errors with that, then marks will be deducted accordingly.

**Submission**

**Note**: For the final submission, you need to complete and submit the files corresponding to the last 2 points and upload the .zip file for the same. The sample submission docs are attached below for your reference.

**Submissions required**

Upload a zip file containing the following:

1. ​​​​A PDF document (**LoadNoSQL.pdf**) containing the commands to load the transactions history data (card\_transactions.csv) in a NoSQL database.
2. A PDF document (**SqoopDataIngestion.pdf**) containing the code used for ingesting data from the RDS server.
3. A PDF document (**CreateNoSQL.pdf**) to create a look-up table with columns specified earlier in the problem statement.
4. Script to calculate the moving average and standard deviation of the last 10 transactions for each card\_id for the data present in Hadoop and NoSQL database. If the total number of transactions for a particular card\_id is less than 10, then calculate the parameters based on the total number of records available for that card\_id. The script should be able to extract and feed the other relevant data (‘postcode’, ‘transaction\_dt’, ‘score’, etc.) for the look-up table along with card\_id and UCL. (**PreAnalysis.pdf**)  
   **Note:** In this pdf provide all the commands to load the lookup table with the relevant data.
5. Screenshots of the execution of the scripts written. The scripts should, after loading the data and creating the look-up table, take the data from the NoSQL database and AWS RDS and perform the relevant analyses as per the rules and should feed the data in the look-up table (**ScriptsExecution.pdf**).
6. Explanation of the solution to the batch layer problem in detail should be provided properly in a document. (**LogicMid.pdf**)  
   **Note: All the sample files have been provided to you as a word document at the end of this segment in a zipped file.**
7. The structure of your directory should be as described ahead. A directory named "**python**" should be created. It should contain a directory named "**src**". The "**src**" directory should have two directories named "**db**" and "**rules**".  The "**src**" directory should have a python file named "**driver.py**" which should be the calling the other files and should be the entry point of your code. The "**rules**" directory should contain a "**rules.py**" file where you write the functions to check for the three rules mention earlier. The "**db**" directory should have the "**geo\_map.py**" and "**dao.py**" shared earlier. The **driver.py**file should contain the code to read the messages from Kafka and call necessary functions from the python files present in the "**rules**"and "**db**"directory to classify the incoming transaction as fraud or genuine.  
   **Note: It is not necessary to follow the above guidelines. Whatever the code you write, you need to properly comment it and also document the exact steps to run the code provided by you.**  
   **Note: It is very important to provide a document to explain the code that you have written for the streaming layer. Also, you need to properly document the steps to execute the same code.**
8. Explanation of the solution to the streaming layer problem in detail should be provided properly in a document. (**LogicFinal.pdf**)  
   **Note: This is a must. You need to clearly explain the code and the steps to run the same should be properly documented.**

**Please make sure that you are not changing any of the file names that have been provided above in parentheses. The code that you are submitting should run at our end without any modifications in the code.**

The following zip file contains the sample documents and also contains the directory structure as explained in **point 7**.