Análisis de regresión logística para clasificación de unidades de transporte de acuerdo con su uso y consumo de combustible

Luis Arturo Rendón

A01703572 – Tecnológico de Monterrey Campus Querétaro

Resumen

Se presenta un estudio realizado a datos obtenidos de las unidades (camiones) pertenecientes a una empresa dedicada al transporte de carga y a la logística. El objetivo es el desarrollo de un modelo capaz de clasificar unidades según su uso y consumo de combustible; facilitando la detección temprana de patrones anómalos facilitando la toma de decisiones, prevención de errores y mejora general de la gestión de la flota y disminución de costos.

Introducción

En la actualidad, la industria del transporte y la logística son parte fundamental del mundo globalizado en el que vivimos, siendo una vértebra vital en la cadena de suministro tanto nacional como internacional. Sin embargo, la eficiencia operativa, especialmente en términos de consumo y robo de combustible por parte de los operadores, se ha convertido en un desafío vital para las empresas debido a su impacto en los costos y en el impacto ambiental.

Usando la regresión logística como metodología principal, se analizarán datos históricos de las unidades para delimitar los datos que permiten clasificarlas de una manera eficiente. El presente reporte presenta (1) la obtención y (2) descripción de los datos, (3) como se ajustaron para (4) el modelo (5) el entrenamiento, (6) uso del modelo y los resultados obtenidos con el mismo y (7) resultados tras usar el framework sklearn en python.

1. Origen de los datos

Los datos pertenecen a DAM, empresa de transporte y logística ubicada en el estado de Querétaro. La empresa cuenta con 10 unidades operando dentro de la república mexicana, principalmente zona centro. La información fue extraída de Cloudfleet, un

programa para la administración y mantenimiento de unidades de transporte.

2. Descripción de los datos

Se descargó de la página un CSV con el historial de cargas de combustible a todas las unidades. El sistema fue adoptado recientemente por la empresa y en consecuencia solo se cuentan con 632 registros comenzando el 1ro de abril del 2024.

2.1. Descripción de las columnas

El CSV consta de 11 columnas en el siguiente orden:

- Nro: Número de carga de combustible
- Vehículo: Identificados del vehículo, del 101 al 110.
- Odómetro: El kilometraje de la unidad.
- Horómetro: La cantidad de horas que la unidad ha estado en operación.
- Fecha: La fecha de la carga de combustible.
- Tanqueo Full (La falta de ortografía viene en el CSV descargado): S si el tanque se llenó o N si el tanque no se llenó por completo.
- Costo por Volumen: El costo en pesos mexicanos por unidad de combustible.
- Cantidad: Cantidad de unidades de combustible.
- Unidad: Unidades de combustible (Litros).
- Costo Total: El costo total de la carga de combustible.
- Tipo: El tipo de combustible (Diesel).

3. ETL (Extraer, Transformar y Cargar)

El CSV descargado es cargado al programa mediante la librería DataFrames.jl del lenguaje de programación Julia, mismo que será utilizado por toda la realización del proyecto.

A partir de este momento nos referiremos al DataFrames con la información del CSV como df para mayor simplicidad.

3.1. Primera inspección

Al desplegar el df y hacer una primera inspección resaltan dos principales conflictos, el primero es que la columna de Horómetro solo presenta 14 registros y el segundo que donde la columna sí tiene registrado un dato, se presentan dos comas después. Esto empuja a todos los demás datos de la fila una columna a la derecha, creando conflicto en estas filas y creando una columna extra con 14 datos mal posicionados.

3.2. Solución de formato

Tomando en cuenta la poca cantidad de datos que se tienen, en vez de eliminar las 14 filas que presentan problemas se diseñó una función para solucionar el conflicto de las comas extra acomodando dichas filas al estándar correcto del documenta. De esta

manera pudimos mantener la integridad de la información intacta.

Una vez solucionado el conflicto con el CSV guardando la información con buen formado en otro documento, procedemos a cargarla de nuevo para ahora sí, manejar el df sin que presente mayores problemas.

3.3. Manejo de datos nulos

En todo el df solo se faltan datos en la columna de Horómetro y en vista de que de los 632 registros solo se encuentran 14, se elimina la columna entera por baja relevancia.

3.4. Por qué se solucionaron los problemas de todas las columnas antes de seleccionar solo las que se iban a usar

Para el modelo se seleccionan tres columnas para trabajar, dos de entrada y dos de salida. Sin embargo, para fines de aprendizaje de ETL y del lenguaje Julia, se decidió manejar todos los datos en el df antes de elegir las columnas a utilizar.

3.5. Manejo de tipos de dato

Todos los tipos de dato a partir de la tercera columna estaban en cadenas de texto para lo que se hicieron las siguientes conversiones:

- Fecha: DateTime
- Tanqueo Full: Int32 (1, 0)
- Costo por Volumen: Float32
- Cantidad: Float32
- Costo Total: Float32

Se usaron Int32 y Float32 para el ahorro de memoria, no se necesitaban de 64.

3.6. Cambio de nombre por columna

Para facilitar el manejo de las columnas en el df y solucionar errores ortográficos se cambiaron los nombres de las columnas. Cabe mencionar que se solucionó el error de Tanqueo Full, pero se eliminaron los acentos de todos los nombres, para facilitar su manejo y escritura.

- Nro.: Numero
- Vehículo: Vehiculo
- Odómetro: Odometro
- Tanqueo Full: Tanque_Lleno
- Costo por Volumen: Costo_Por_Volumen
- Cant.: Cantidad
- Costo Total: Costo_Total

3.7. Escalamiento

Debido a la diferencia de escala entre variables se aplicó la fórmula de normalización. Como se ve en las gráficas siguientes los datos no presentan ningún comportamiento o distribución normal. Sin embargo, si se encuentran muy alejados en escala, desde 24 hasta 1,000,000. Por ende, se decidió utilizar la fórmula de normalización y no la de estandarización. De esta forma pudimos acotar los valores entre 0 y 1.

3.8. Gráficas

Para la búsqueda de patrones en los datos utilicé principalmente ScatterPlots y una gráfica de barras.

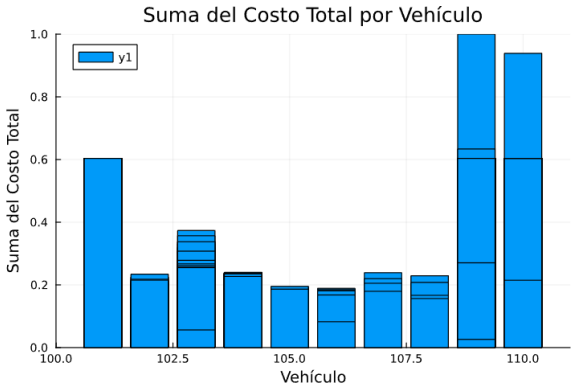


Ilustración 1: Gráfica de barras, Vehículos, Costo Total

En esta primera gráfica lo que hice fue comparar los vehículos con el costo total de la recarga de gasolina. Esta primera gráfica no la planifiqué bien ni le di un propósito específico, fue más bien para comprender las gráficas en el lenguaje de Julia. Cabe mencionar que ya de entrada se ve un patrón en las unidades, con algunas que consumen mucho más combustible que otras.

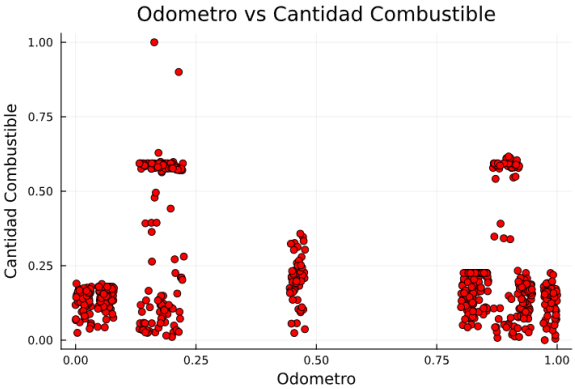


Ilustración 2: Scatter, Odómetro, Cantidad de Combustible

Ahora sí quise encontrar patrones en la información, al ser cargas de combustible a diferentes camiones quise primero probar si un mayor kilometraje en una unidad aumenta el consumo de combustible. Sin embargo, se ve un patrón similar a la gráfica anterior, se ven agrupaciones de puntos en distintas áreas denotando clases (Vehículos).

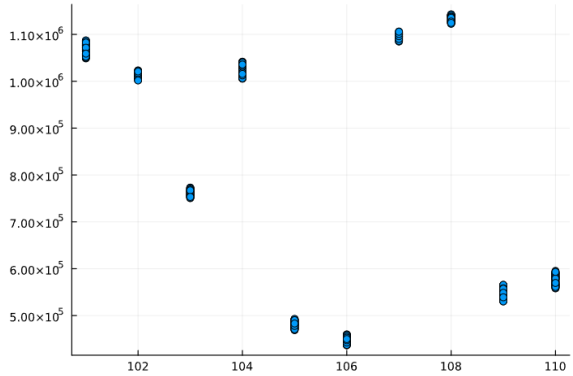


Ilustración 3: Scatter, Vehículos, Odometro

Esta siguiente gráfica muestra a las unidades con el odómetro que presentaban, la verdad viendo la gráfica ya a detalle y reflexionando aquí mismo sobre ella no sé en qué estaba pensando, es obvio que iban a ser agrupaciones ya que cada unidad tiene su kilometraje.

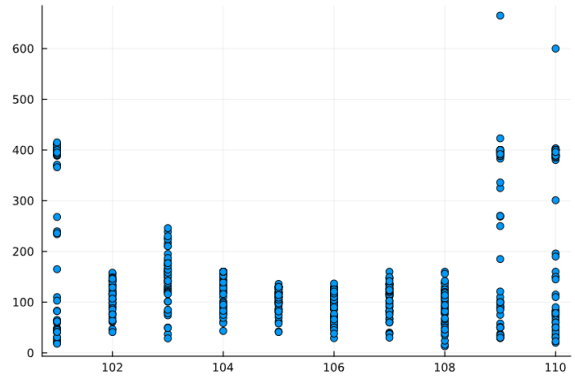


Ilustración 4: Scatter, Vehículos, Cantidad de combustible

Esta cuarta gráfica muestra los vehículos con sus respectivas cantidades de combustible en la carga, se puede ver una gráfica prácticamente igual a la de barras del principio. Sé por la empresa que cuentan con dos camiones de carga grandes, se ve claramente en la información como la unidad 109 y la 110, ya que usan la mayor cantidad de combustible por un margen notorio.

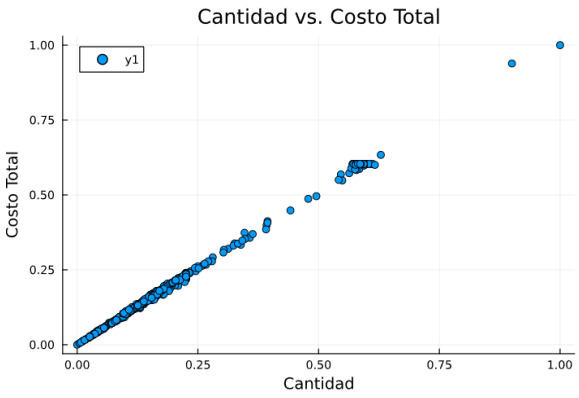


Ilustración 5: Scatter, Cantidad de combustible, Costo total

La única gráfica que presentó una correlación sumamente fuerte fue la de Cantidad de Combustible y Costo Total. Pero al ser el precio de la gasolina por litro, es un comportamiento esperado que causa poco interés y sin mucha utilidad.

En vista de la dificultad para la aplicación de un modelo de regresión lineal, se decidió usar un modelo de regresión logística para la clasificación de las unidades con respecto a su odómetro y a la cantidad de combustible.

Y en vista de que tengo evidentes agrupaciones de datos por vehículo decidí hacer un modelo que predica el vehículo en base al odómetro y a la cantidad de combustible.

3.9. Shuffle

Para asegurar que los datos estén unánimemente repartidos se revolvieron las filas del df original.

3.10. Selección de información

En un nuevo df se guardaron las columnas de entrada que serían el odómetro y la cantidad de combustible. Y en otro df que llamaremos y, se guardaron el número de las unidades. Se eligieron estas dos variables gracias a una consulta que se le hizo a mi suegro para que me las explicara. El Odómetro representa el kilometraje de dicha

unidad al momento de la carga de combustible, variable que permite clasificar con facilidad a los vehículos debido a que todos tienen valores diferentes y representativos. Además, se eligió la cantidad de combustible porque esto separa las unidades por su tamaño, especialmente a los tractores grandes de los camiones que se usan dentro de CDMX. Usando estas dos variables el chiste es ver que unidad es la que cargó gasolina.

3.11. OneHotEncoding

Debido a que se tienen 10 unidades por clasificar y el modelo a implementar solo consta de una neurona de salida, este solo podrá analizar y diferencias a una unidad de las demás. Debido a esto se tomó la decisión de entrenar múltiples modelos, 1 por cada unidad. Debido a esto a y se le aplicó OneHotEncoding para separar las unidades en otro df donde cada unidad tiene su columna con 1 si es su fila correspondiente o 0 si no lo es.

4. El modelo

La siguiente imagen muestra la idea general del modelo a utilizar, de esta manera se pudo entender y programar con mucha mayor facilidad.

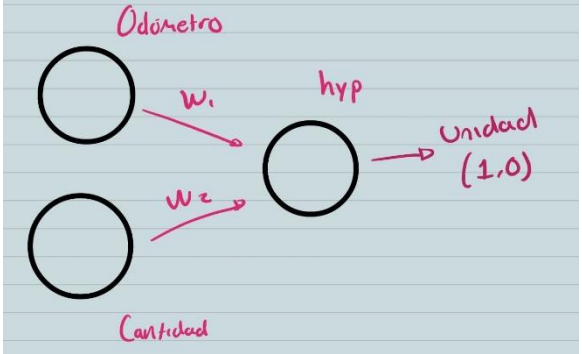


Ilustración 6: Boceto hecho a mano del modelo a implementar

4.1. La hipótesis

La función que utiliza la neurona es la siguiente:

$$y = x1 * w1 + x2 * w2 + b$$

Las dos x son las columnas de entrada y las w son los respectivos parámetros o pesos de cada una. Los pesos (w) son los parámetros que iremos ajustando conforme el modelo aprenda.

Por último, la b es el “bias”, el cual es un parámetro que le ayuda al modelo a ajustarse con mayor rapidez a la data en uso. Este no es alterado durante el uso del modelo.

4.2. Gradiente descendiente

Esta función es la que realiza el trabajo de la neurona, haciendo una predicción usando la hipótesis para cada línea del df.

Con la hipótesis y el valor esperado de la fila se calcula el error de esta.

4.2.1. Calcular el ajuste de los parámetros

Para calcular el ajuste de las w se utiliza la siguiente fórmula:

$$-(y * hyp) * hyp * (1 - hyp) * x$$

- y: es el resultado esperado
- hyp: es el resultado obtenido
- x: es el input con respecto a la w a ajustar

El resultado de dicha ecuación multiplica por el valor de alpa, el cual también es conocido como la taza de aprendizaje. Este valor le ayuda al modelo a aprender más rápido o más lento, dependiendo el problema al que se enfrente y la velocidad deseada por el usuario. Por último, el resultado de dicha multiplicación se le resta al parámetro original.

$$wi = w - a * resultadoPrevio$$

5. Entrenamiento

5.1. Separación de data

Se decidió separar el df y la y en 3 partes. La primera es para entrenamiento y se le designó el 80 % de los datos, porque tenemos una cantidad reducida de ellos y necesitamos el mayor número para asegurar un buen entrenamiento.

Después se le asignó un 10% a validación, para asegurar la integridad del modelo antes de probarlo oficialmente. Y por último se le asignó 10% a las pruebas para ver los resultados y porcentajes de eficiencia del modelo.

5.2. Entrenamiento con trainData

El modelo corrió 600 épocas con el df de entrenamiento con el cual el error se movió de 0.61022 a 0.8168. Al principio causó confusión el hecho de que el error haya aumentado, pero al momento de graficarlo y validar el modelo me di cuenta de que mientras más cercano a 1 el modelo era más preciso. Por ende, 0.81 presentaba una eficacia aproximada a 81% con los datos de entrenamiento.

Cabe mencionar que las 2 terminaron de esta manera:

- w1: 7.8412
- w2: 2.8352

Es interesante como el modelo cargó la mayor parte de los ajustes al peso 1, más adelante hablaré del tema.

5.3. Gráfica del error

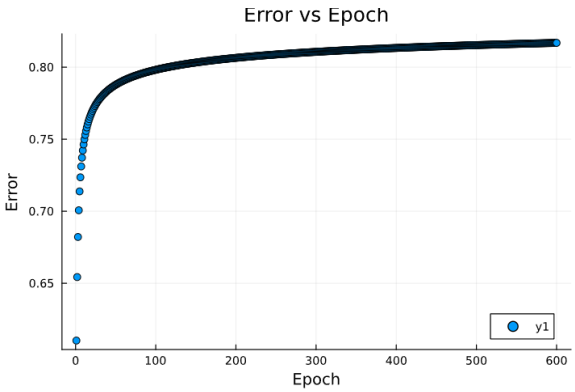


Ilustración 7: Scatter, Epoch, Error

En la gráfica anterior se ve lo mencionado, el error aumentó hasta llegar a unos 82.

6. Uso y pruebas

La validación mostró un resultado de éxito del 87% lo cual me permitió pasar al conjunto de prueba.

Con los datos que se tenían guardados en test y el modelo con los parámetros mencionados anteriormente se obtuvo una precisión del 92%.

6.1. Revisión de los resultados obtenidos

> Para corroborar los resultados y los porcentajes obtenidos se decidió usar una matriz de confusión.
>
> Quiero decir en este momento, de manera extra-profesional, que dolió enormemente el resultado de la matriz.

	0	1
0	0	5
1	59	0

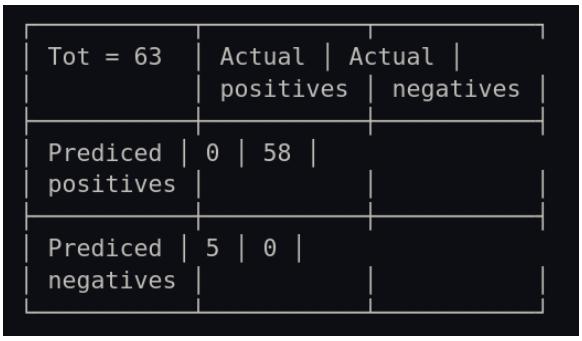
Gracias al haber realizado la tabla, me di cuenta de que el modelo estaba apostando siempre al 1 cuando tenía que ser 0. De esta manera descubrí 3 cosas:

- La primera: Por eso el peso de w1 está tan cargado hacia un lado, lo está favoreciendo mucho para obtener siempre un resultado de 1.
- La segunda: Por eso el error va hacia arriba, siempre que sabe que será 0, pone 1. Entonces está buscando equivocarse siempre (siempre atinarle).
- La tercera: Como ve que casi siempre el resultado es 0, prefiere solo apostarle al 0 para asegurar todo lo posible el resultado esperado.

Esto muestra que el modelo está underfitting, no es capaz de predecir correctamente los resultados esperados, solamente le apuesta al número que más aparece y ya.

6.2. Resultados validación

De la validación obtuve también la matriz de confusión, la cual dio resultados casi iguales:



Al igual que en la matriz de confusión del train, se ve que el modelo busca lo inverso, como si buscara aumentar el error todo lo posible en vez de disminuirlo. La parte inversa no me molesta, solamente lo el underfitting es problemático, como en el train, el modelo solo le apuesta a un valor y ya.

```
Dict{String, Real} with 8 entries:
  "precision@fpr0.05"          => 1.0
  "recall@fpr0.05"             => 0.0
  "accuracy@fpr0.05"           => 0.920635
  "au_prcurve"                 => 0.0396825
  "samples"                    => 63
  "true negative rate@fpr0.05" => 1.0
  "au_roccurve"                => 0.0
  "prevalence"                 => 0.0793651
```

Podemos ver un accuracy del 92%.

6.3. Resultados test

Del test se obtuvo una matriz de confusión de nuevo parecida a las anteriores.

Tot = 64	Actual positives	Actual negatives
Prediced positives	0	54
Prediced negatives	10	0

Donde el modelo busca equivocarse propósito invirtiendo todo. Para esto, se sigue viendo el mismo underfitting, donde el modelo aprende que son tan pocos los valores que tiene que predecir correctos, que es mejor apostarle a siempre tirar 1, de esta manera se mantiene en un muy buen porcentaje de éxito. No es capáz de ajustarse para predecir cuando va a aparecer la unidad deseada.

```
Dict{String, Real} with 8 entries:
  "precision@fpr0.05"          => 1.0
  "recall@fpr0.05"             => 0.0
  "accuracy@fpr0.05"           => 0.84375
  "au_prcurve"                 => 0.078125
  "samples"                    => 64
  "true negative rate@fpr0.05" => 1.0
  "au_roccurve"                => 0.0
  "prevalence"                 => 0.15625
```

Podemos ver un accuracy del 84% en el test.

6.4. Conclusión de mi modelo

Después de utilizar mi modelo varias veces para clasificar a una unidad me di cuenta de dos cosas. Primero, como el one-hot encoding me pone una columna por cada vehículo cuando quiero clasificar uno, casi todos los registros salen en 0 y unos pocos en 1. Esto causa que el modelo aprenda a siempre poner el valor erróneo, de igual manera con eso sale con porcentajes de éxito de 90% para arriba.

A primera vista parece que el modelo funciona bien ya que presenta resultados similares entre el train, el test y el validation con poca varianza. Pero esto se desmiente con las matrices de confusión y se revela la técnica que está tomando mi modelo.

También se me hizo interesante que el modelo se da cuenta velozmente de cuál variable es la que más peso tiene con respecto a qué unidad es. Ahora que lo pienso bien, es algo obvio ya que para identificar una unidad es mucho más fácil hacerlo mediante el odómetro (kilometraje) ya que todas tienen uno diferente, la cantidad no varía tanto entre los vehículos. Debido a esto el modelo se da toda la importancia al parámetro del odómetro y le da poca importancia al otro.

Para manejar este tipo de clasificación, con valores tan representativos como el kilometraje queda mucho mejor la clasificación por árboles. De hecho, por eso decidí usar RandomForestClassifier con el framework, lo cual me dio resultados excelentes.

7. Scikit-Learn (Random Forest Classifier con framework)

Como teníamos que resolver el problema con un framework, primero sopesé la idea de hacer una regresión logística tal cual como la mía para poder compararlas. Pero en las últimas sesiones vimos el método de árboles y bosques, el cual se me hizo el IDEAL para el problema que intentaba solucionar. Como tengo datos tan notoriamente diferente entre los distintos vehículos o categorías le aposté a que un árbol que tomara decisiones en base a las diferencias entre los vehículos lo solucionaría con extrema facilidad y así fue.

7.1. ETL

No tengo mucho que comentar en este modelo, lo realicé bastante fácil (Hermosos frameworks).

Para el ETL tomé el CSV que ya había impreso el otro modelo, el que ya tenía solucionado el tema de las columnas y las comas. Además, metí solo las columnas deseadas (Las mismas que en el modelo pasado). En estas columnas no tengo valores nulos y también aprovecho para convertir los datos de strings a valores numéricos.

A continuación, normalizo los datos para poder ingresarlos al modelo, además para no tener valores tan extremos como las diferencias entre la cantidad y el odómetro. Cambio los nombres de las columnas por mera comodidad mía y divido en train y test, 90% y 10% respectivamente.

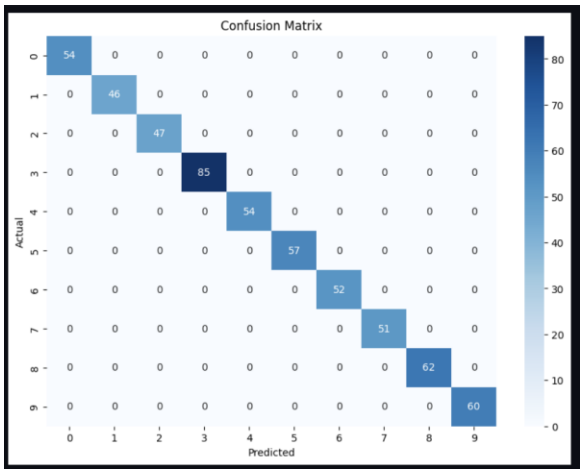
7.2. Entrenando y usando el modelo

Por último, llamo al modelo RandomForestClassifier() y lo entreno. Ya con el modelo entrenado corro el test obteniendo el hermosísimo resultado de 0.88 de precisión. Mi hipótesis con el framework era correcta, random forest era el ideal y me dio un excelente resultado de 88% de precisión.

De hecho, fue la implementación tan agradable y el resultado tan satisfactorio que me di cuenta de que una regresión logística era algo intenso en exceso, más para categorías que presentan valores tan diferentes en algunas áreas.

7.3. Train

La matriz de confusión del entrenamiento muestra casi puros aciertos.

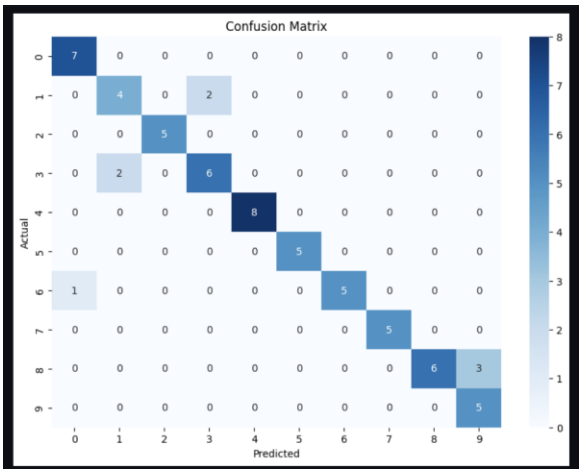


El vehículo número 3 parece ser el que más aparece en el data-set.

No aparece ningún error en las predicciones durante el train, esto me hace pensar un poco en overfitting incluso usando el framework. Eso o los datos son sumamente fáciles de manejar y el modelo los domina muy bien... pues eso es overfitting no? Jajaja.

7.4. Test

Ahora el momento importante, durante el test modelo si presenta errores. Como se puede ver abajo tiene 3 errores con la unidad 8, 2 errores con la unidad 1, 1 error con la unidad 5 y 2 errores con la unidad 3. Al final obtenemos un accuracy del 88% como mencionamos previamente. Número que me tiene muy satisfecho.



Al final el overfitting que me tenía tan preocupado no afectó a los resultados del test como me temía. Predice eficientemente los datos que no conoce.

8. Conclusión de mi modelo

Mi modelo en Julia presenta underfitting. Ya de entrada tengo que dividir mi columna de vehículos en 10 columnas, una para cada vehículo con 1 donde aparece y 0 donde no aparece. Con eso tengo columnas muy vacías con pocos 1. Luego si le sumamos que programé una regresión logística la cual puede predecir entre dos categorías, me metí en un gran problema.

Mi modelo solo puede clasificar entre una unidad y las que no son esa unidad, toma una de las columnas y la utiliza para saber cuándo una unidad si queda bien con los datos y cuando no. El siguiente conflicto fue que, de las dos variables elegidas, me di cuentas hasta muy tarde que con solo una de ellas puedes predecir qué unidad es. Como cada unidad tiene un kilometraje tan particular, con este dato basta para

tomar decisiones y saber qué unidad es. Me di cuenta de esto cuando mi modelo le ponía mucho peso a un parámetro y medio abandonaba el otro.

Y por último, al set tantos 0 y tan poquitos 1, mi modelo salió sumamente underfitteado. Como puede solo ajustar 2 parámetros y solo tiene una neurona no le quedó más que aprender a siempre tirar 0 (bueno mi modelo tira 1, pero ya expliqué anteriormente que lo hace todo al revéz) y con eso ya consigue porcentajes de éxito del 90%, pero si le presentara datos donde la unidad sí haga presencia, o si en verdad quisiera ubicarla, no tendría manera de cómo hacerlo.

9. Conclusión del framework

Después de la explicación que vimos en clase de los árboles de decisiones, me di cuenta de que esta sería la solución perfecta para mi problema. De hecho, decidí no hacer un modelo equivalente al mío con el framework, decidí usar árboles de decisión.

Como ya presenté antes, los resultados fueron los que me esperaba, tengo un porcentaje de éxito en el test del 88% lo cual es asombroso. No solo eso, el modelo usa las 10 unidades sin problema alguno y las clasifica como si nada (duele, duele el orgullo).

La primera matriz de confusión me espantó, como no salió error creí que el modelo saldría overfitting y que no podría con valores ajenos al test, pero con el test me di cuenta de que puede hacer las predicciones de manera muy eficiente y atinada.