Proyecto 2

1st Barrios, Alonso
Universidad de Tecnología e Ingeniería
Ciencia de la Computación
Lima, Perú
alonso.barrios@utec.edu.pe

2nd Rodriguez, Mauricio
Universidad de Tecnología e Ingeniería
Ciencia de la Computación
Lima, Perú
mauricio.rodriguez@utec.edu.pe

3rd Tenazoa, Renzo
Universidad de Tecnología e Ingeniería
Ciencia de la Computación
Lima, Perú
renzo.tenazoa@utec.edu.pe

***Abstract*—Este proyecto está desarrollado para poner a prueba lo aprendido acerca de los modelos de clasificación de regresión logística, *svm*, árbol de decisión y *knn*. El objetivo del proyecto es clasificar un *set* de datos de *testing* haciendo uso de los modelos mencionados y analizar las distintas gráficas obtenidas por los errores de entrenamiento y validación y *testing* para así observar cual modelo tiene un mejor *accuracy*.**

***Index Terms*—clasificación, error, underfitting, overfitting**

I. Introducción

Con el objetivo de poner a prueba los modelos de clasificación se utilizó el *dataset* Gender Classification, en cual consta de 5001 intancias con 7 atributos cada una que representan distintas características de las personas. Se buscó clasificar el campo denominado como *gender*, el cual representa el género de la persona que se desea clasificar según sus características. Posteriormente, en el proceso de entrenamiento del modelo se obtendrán distintas gráficas las cuales representan al error de entrenamiento, validación y/o *testing* del modelo. Se usaron estas gráficas para visualizar si se obtuvo *underfiting* u *overfiting* en el modelo de clasificación.

II. Explicación

Para la implementación de los modelos de clasificación se usaron las siguientes funciones generales:

- *passdata()*: Se encarga de recolectar los datos del *dataset* para almacenarlos en listas.
- *normalize()*: Se encarga de normalizar los datos ya que la diferencia entre estos llega a ser muy grande. Cabe resaltar que se normalizó tanto el x e y, además en x no se normalizaron las columnas *month* y *day*.
- *shuffle()*: Se encarga de "chocolatear" el *set* de datos tanto de los arreglos x como y.
- *sliptData()*: Se encarga se separar la data en *training*, *validation* y *test* según los porcentajes pasados como parámetros.
- *getAccuracy()*: Se obtiene el porcentaje de *accuracy* según la matriz de confusión pasada como parámetro.
- *getError()*: Se obtiene el porcentaje de *error* según la matriz de confusión pasada como parámetro.
- *predict()*: Se encarga de retornar los valores y predecidos usando los parámetros ya entrenados y los valores x.
- *real()*: Se encarga de retornar los y del *dataset*.

También se hizo uso de las siguientes funciones específicas para cada modelo:

- Regresión logística:
  - *s()*: Es el resultado que se obtiene de la predicción cada vez volviéndose más preciso.
  - *derivate()*: Se encarga de derivar las funciones necesarias.
  - *error()*: Se encarga de calcular el error de cada fase de entrenamiento.
  - *update()*: Se encarga de actualizar los valores para que cada fase de entrenamiento sea más precisa.
  - *testing()*: Se encarga de calcular los valores de error de la parte de *testing* una vez se tiene los parámetros entrenados.
  - *train()*: Es la función principal en la que se empieza a entrenar el modelo de clasificación.
- *SVM*:
  - *hypothesis()*: Es el resultado que se obtiene de la predicción cada vez volviéndose más preciso.
  - *derivate()*: Se encarga de derivar las funciones necesarias.
  - *error()*: Se encarga de calcular el error de cada fase de entrenamiento.
  - *testing()*: Se encarga de calcular los valores de error de la parte de *testing* una vez se tiene los parámetros entrenados.
  - *train()*: Es la función principal en la que se empieza a entrenar el modelo de clasificación.
- Árbol de decisión:
  - *CSVReader::read()*: Se encarga de *parsear* los datos que se encuentran dentro de un archivo CSV.
  - *build_tree()*: Se encarga de la construcción del árbol de decisión para poder así poder realizar las predicciones.
  - *get_column()*: Se encarga de obtener todos los valores de una columna de la matriz de datos, es decir, todos los valores de un atributo.
  - *split()*: Se encarga de dividir los datos en dos partes.
  - *get_best_split()*: Se encarga de obtener el mejor *split* con un *information gain* menor.
  - *gini_gain()*: Se encarga de devolver el indicador que nos permitirá encontrar el mejor *split*
  - *calculate_leaf_value()*: Se encarga de calcular el valor final de la clasificación en el último nodo.
  - *generatePDF()*: Se encarga de mostrar el árbol de

decisión en formato PDF (funciona el linux).
  - *predict()*: Dado una matriz de datos, se encarga de predecir la clasificación en base a esas características.
- *KNN*:
  - *insertAll()*: Se encarga de insertar todo el *dataset* al *RTree*, esta función se utilizaría después de hacer el *testing*.
  - *clear()*: Se encarga de eliminar los archivos creados por la librería *rtree* utilizada con el fin de "resetear" el árbol.
  - *countNeighbors()*: Se encarga de hacer el conteo de cuántas veces se repiten los k vecinos más cercanos.
  - *knn()*: Se encarga de obtener los k vecinos más cercanos y retorna el *y* que más se repite.
  - *kFoldCrossValidation()*: Se encarga de graficar los resultados obtenidos usando el teorema del límite central; además, hace el uso del método *K-Folds cross-validator* para hacer pruebas en el modelo.

Con el fin de mejorar los modelos de clasificación de regresión logística y *svm* se dividió el *dataset* de manera aleatoria en 3 partes:

- *train*: Es el 70% de los datos con los que se entrenó al modelo.
- *validation*: Es el 20% de los datos con los que se valida el modelo.
- *test*: Es el 10% de los datos con los que se pone a prueba el modelo.

Mientras que para el *knn* se usó el método *K-Folds cross-validator*, el cual hace uso solo de *training* y *testing*. Y para el árbol de decisión se separó la *data* en un 80% para armar el árbol (*training*) y un 20% para *testing*.

III. Experimentos

A. Regresión Logística

Los siguientes resultados se obtuvieron normalizando las columnas 2 y 3 del *dataset* y con un *alpha* igual a 0.1:

- Matriz de confusión

Train

1659	83
40	1718

Validation

478	24
11	487

Testing

250	7
6	238

- *Accuracy*

Train	96.48571428571428
Validation	96.5
Testing	97.40518962075848

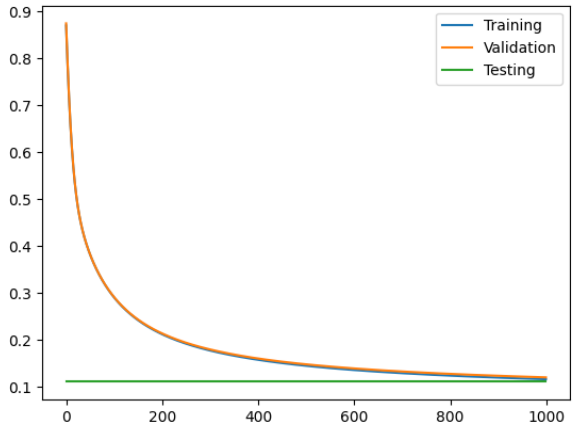

Fig. 1. Regresión Logística - α: 0.1

- Gráfica de errores

En la gráfica se puede visualizar y confirmar el cómo los errores de *training* y de *validation* del modelo empiezan siendo altos y van disminuyendo, de forma que se genera una curva similar a una esperada. También se visualiza que la diferencia entre estos errores aplicado a los 2 segmentos es mínima en todo momento, siendo casi imperceptible la diferencia entre las curvas. Esto señala la ausencia de *overfitting* y *underfitting*, indicándonos que el modelo se desarrolló de manera óptima. Además, la gráfica de *testing* nos confirma que el modelo fue entrenado de forma satisfactoria.

Para llegar a estos resultados óptimos, se hicieron una serie de pruebas:

1) ***Alpha*:** La primera prueba fue comprobar el correcto funcionamiento del algoritmo con diferentes *alphas*.

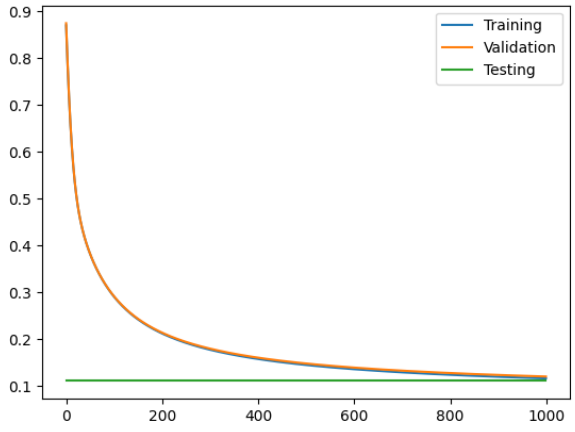

Fig. 2. Gráfico de errores con *alpha* = 0.1

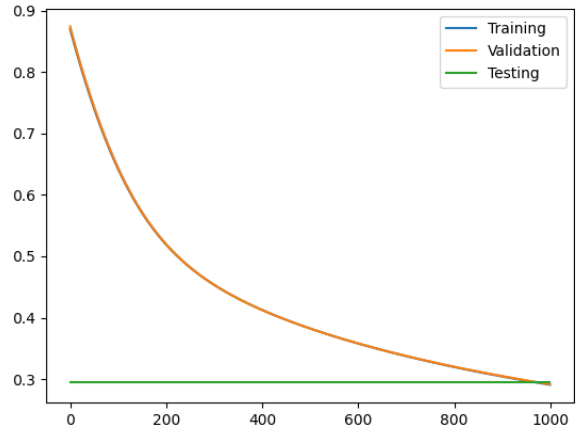

Fig. 3. Gráfico de errores con *alpha* = 0.01

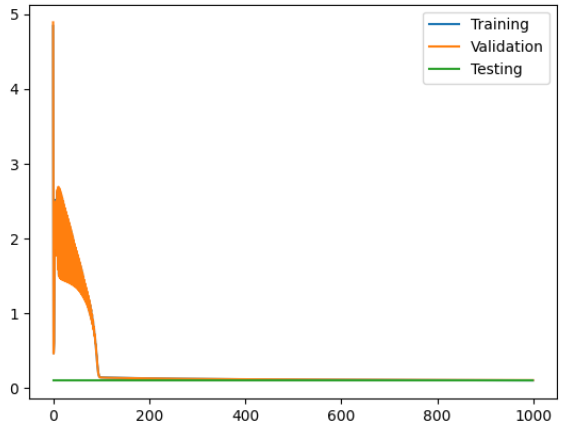

Fig. 5. Gráfico de errores con datos no normalizados y *alpha* = 0.1

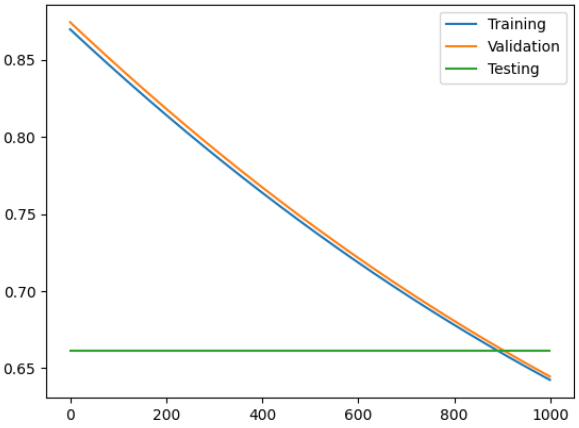

Fig. 4. Gráfico de errores con *alpha* = 0.001

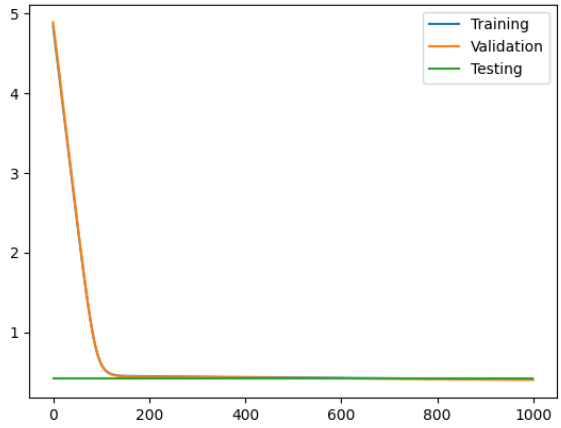

Fig. 6. Gráfico de errores con datos no normalizados y *alpha* = 0.001

Dada las gráficas podemos concluir que con todas se llega a un mínimo óptimo, la diferencia es el tiempo que tomará en llegar a ser óptimo. Esto se debe a que al disminuir el *alpha* este debe de tardarse más, ya que tendrá que dar más saltos para llegar al mínimo óptimo.

2) **Normalización:** La siguiente prueba que se realizó fue la de no normalizar ningún dato.

Dada las gráficas pudimos ver que no hay presencia de *overfitting* ni *underfitting*, sin embargo, cuando los datos están normalizado nos da un mejor *accuracy*.

B. Support Vector Machine

Los siguientes resultados se obtuvieron normalizando las columnas 2 y 3 del *dataset* , con un *alpha* igual a 0.001 y con una constante de C igual a 3:

- Matriz de confusión

Train

1673	69
46	1712

Validation

476	26
11	487

Testing

247	10
4	240

- *Accuracy*

Train	96.71428571428572
Validation	96.3
Testing	97.2055888223553

- Gráfica de errores

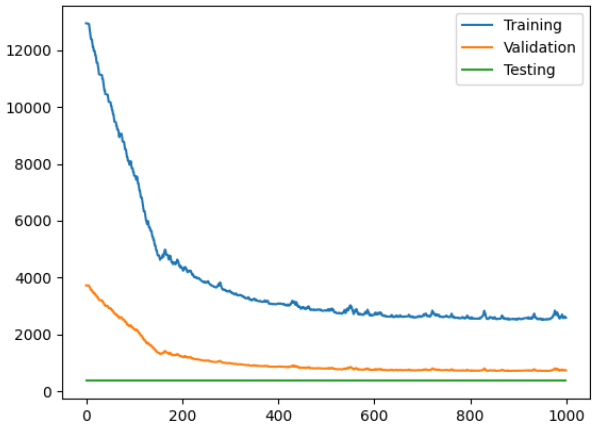

Fig. 7. Gráfico de errores con C = 3 y *alpha* = 0.001

En la gráfica se puede visualizar y confirmar el cómo los errores de *training* y de *validation* tienen un comportamiento similar, sin embargo, la gráfica te *training* está por encima de *validation* lo cual nos indica que nuestro modelo funciona mejor con datos nuevos. Además, vemos que los errores siguen siendo altos pero esto contradice a los resultados del *accuracy* los cuales presentan un buen porcentaje.

Para llegar a estos resultados, se hicieron una serie de pruebas:

1) ***C*:** La primera prueba fue comprobar el correcto funcionamiento del algoritmo con diferentes constantes y un *alpha* igual a 0.001.

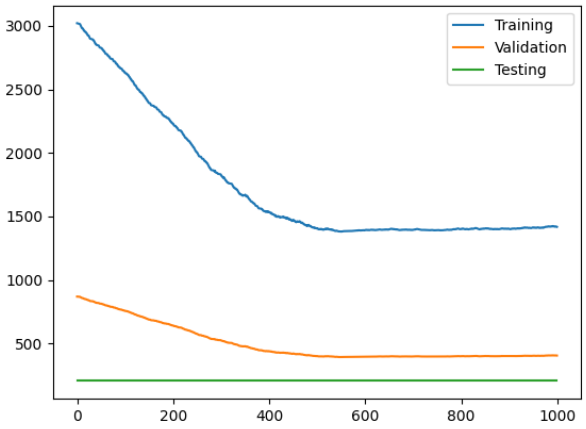

Fig. 8. Gráfico de errores con C = 0.7

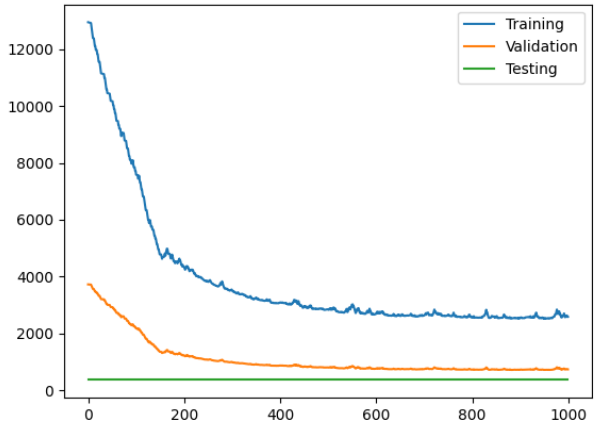

Fig. 9. Gráfico de errores con C = 3

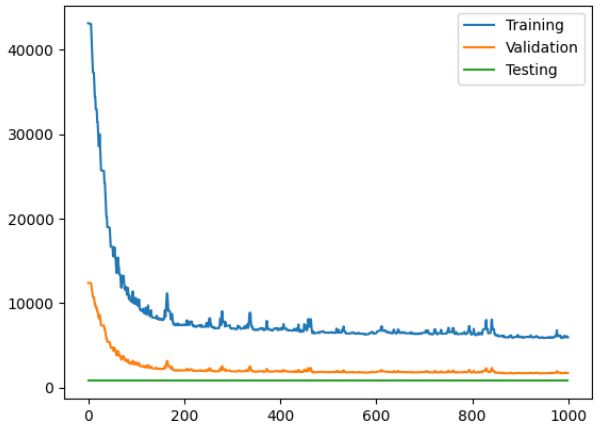

Fig. 10. Gráfico de errores con C = 10

Dada las gráficas podemos observar que cuando la constante es 0.7 presenta *overffiting*, mientras que el resto no. Además, vemos que en todas las gráficas nos da un error alto, sin embargo, los *accuracy* siguen siendo buenos (pasando el 90%).

2) ***Alpha*:** La siguiente prueba fue comprobar el correcto funcionamiento del algoritmo con diferentes *alphas* y con una constante igual a 3.

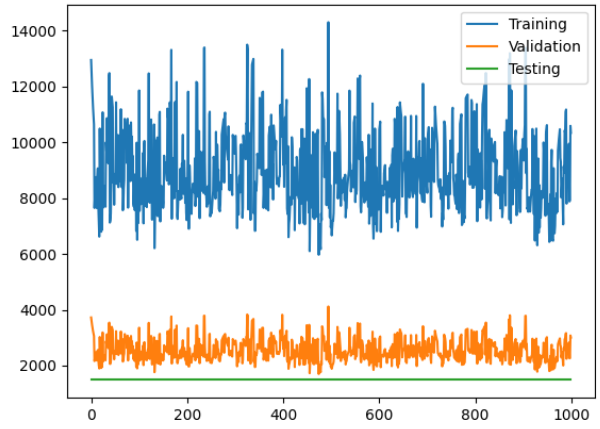

Fig. 11. Gráfico de errores con *alpha* = 0.1

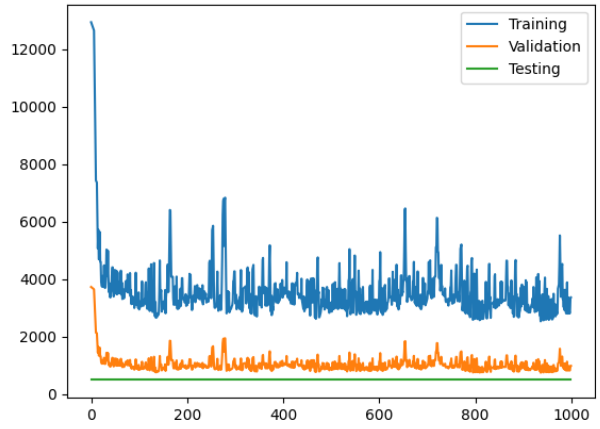

Fig. 12. Gráfico de errores con *alpha* = 0.01

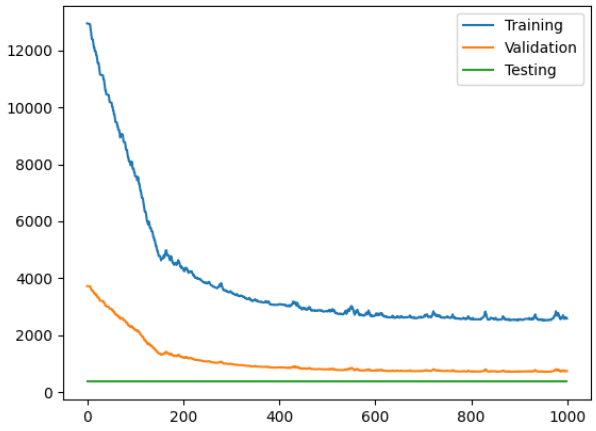

Fig. 13. Gráfico de errores con *alpha* = 0.001

Dada las gráficas podemos observar que mientras el *alpha* sea más grande, las gráficas presentarán más "ruido", además de un mayor error.

3) **Normalización:** La siguiente prueba que se realizó fue la de no normalizar ningún dato y con una constante igual a 3.

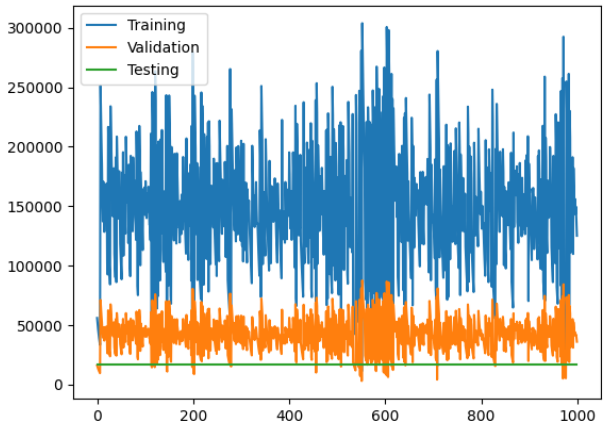

Fig. 14. Gráfico de errores con datos no normalizados y *alpha* = 0.1

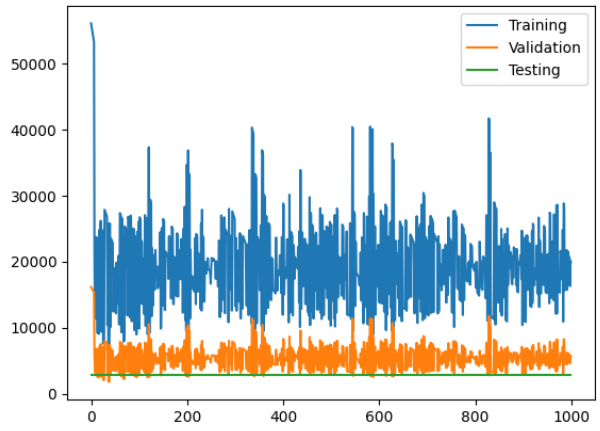

Fig. 15. Gráfico de errores con datos no normalizados y *alpha* = 0.01

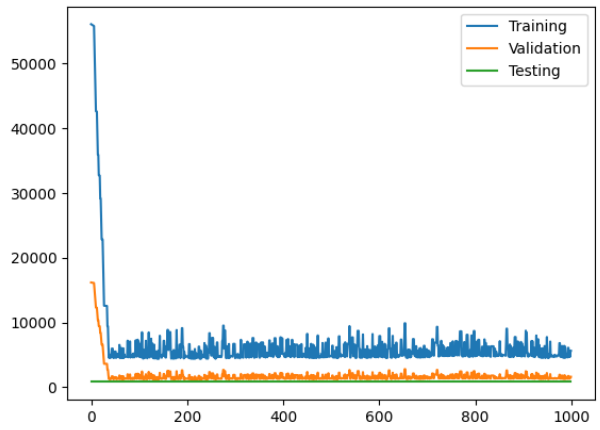

Fig. 16. Gráfico de errores con datos no normalizados y *alpha* = 0.001

Dada las gráficas pudimos ver al no normalizar la data, las gráficas presentan mayor "ruido". Además, hay que resaltar que a pesar de que la última gráfica tiene menor "ruido" y un error aparentemente bajo, el *accuracy* que nos retorna con esos parámetros es bajo, aproximadamente 80%.

C. Árbol de decisión

Para el árbol de decisión se realizó una partición de los datos randomizados, de forma que el 80% de los datos se utilizaran para el entrenamiento del arbol y el 20% para validar su correctitud mediante el testeo correspondiente. Se parametrizó el árbol de forma que la áltura máxima que pueda crear el árbol sea 3. De esta forma, con un *seed* para hacer el *shuffle* se obtuvo.

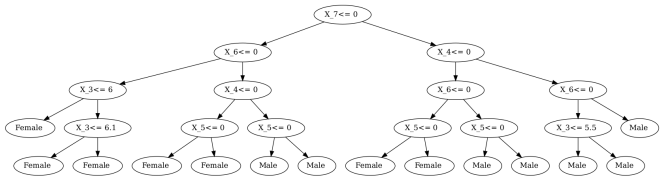

Fig. 17. Decision tree con altura 3

- *Accuracy*

Train	98.7531
Testing	95.2619

- *Gráfica de error*

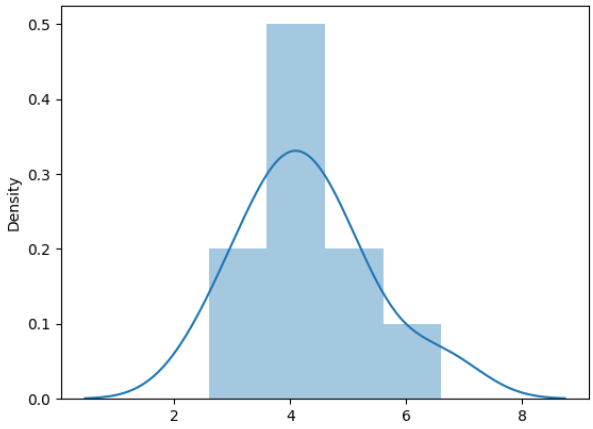

Fig. 18. Decision Tree - Errores

Para el árbol que se muestra se obtuvo un *accuracy* bastante alto, con un 98.75 con los datos de entrenamiento y un 95.26 con los datos de testeo

D. K-nearest neighbors

Para observar los resultados de los experimentos de este modelo se utilzó el método de *K-Folds cross-validator* con el fin de poder visualizar gráficamente los errores que nos retornaba el modelo con el teorema del límite central.
Para ello se usaron como parámetros 10 *folds* y un k que iba desde 1 hasta 40 vecinos. Obteniendo la siguiente gráfica:

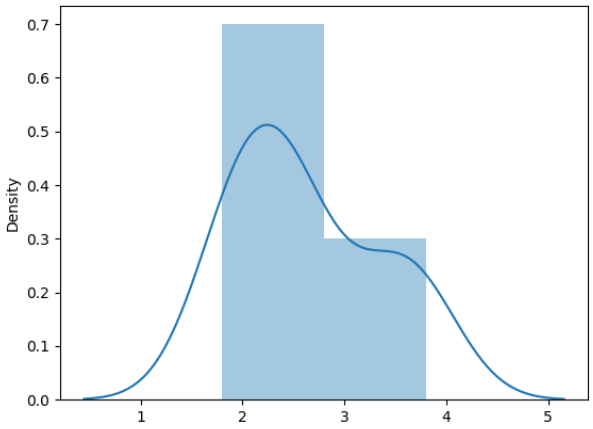

Fig. 19. KNN - k:25 - promedio: 2.64

Este gráfico es el resultado de los parámetros: $k = 25$ y $folds = 10$, siendo este el que menor error daba, incluso mejor que con k vecinos mayores a 25. Por ende podemos concluir que este sería el k ideal para este *dataset*. Obteniendo como resultado un *accuracy* de 97.36. Además, observamos que la gráfica presenta poca varianza y *bias*, lo cual nos indica que no presenta *overfitting* ni *underfitting*.

CONCLUSIONES

- Se aplicaron los conocimientos de modelos de clasificación aprendidos en clase de forma satisfactoria.
- En los experimentos pudimos observar en las gráficas los diferentes comportamientos según los parámetros que se colocaban.
- Tanto de forma visual como numérica se observa que se obtuvieron buenos resultados, ya que las gráficas no presentaban *overfitting* ni *underfitting* y se obtuvo un *accuracy* alto.
- De acuerdo a lo observado los modelos que mejor rendimiento nos han otorgado en cuanto a precisión son los de regresión logística y *support vector machine*.

REPOSITORIO

Enlace al repositorio de Github:
https://github.com/alonso804/ia-project-2

REFERENCES

[1] Montero, R. (2016). *Modelo de regresión lineal múltiple*. Universidad de Granada. España.
[2] Thavanani, S. (2019, 13 diciembre). The Derivative of Cost Function for Logistic Regression. Medium. Recuperado de: https://medium.com/analytics-vidhya/derivative-of-log-loss-function-for-logistic-regression-9b832f025c2d.
[3] Skiena, S. (2017). The data science design manual. Springer.