Proyecto 2

1st Ian Augusto Cortez Gorbalan
ian.cortez@utec.edu.pe
970895554

2nd Plinio Matías Avendaño Vargas
plinio.avendano@utec.edu.pe
927144823

I. Introducción

Este proyecto busca emplear los algoritmos de clasificación supervisada de Regresión Logística, *Support Vector Machine*, Árboles de decisión y *K-Nearest Neighbours*. Se busca una hacer predicciones usando un conjunto de datos reales con características tomadas de hombres y mujeres. Este *dataset* tiene 7 características que definen rostros de hombres y mujeres y una variable que indica si estas características representan a un hombre o a una mujer. El objetivo principal es emplear estos modelos para que sean capaces de adaptarse apropiadamente al *dataset*. Asimismo, como objetivo secundario, se busca determinar cuál de los modelos empleados para clasificación es más apto para el conjunto de datos que se esta utilizando y determinar las características para cada modelo que reduzcan el error para mejorar la eficiencia del modelo.

II. Explicación del modelo empleado

A. Regresión logística

La regresión logística es un modelo de clasificación supervisado que permite separar resultados en dos grupos. Esto se logra al utilizar un hiper-plano ($w^T * x + b$) que mejor separe los dos grupos en base a T características.

Obtener el hiper-plano no nos permite determinar que datos pertenecen a alguno de los dos grupos. Por lo tanto, se emplea la ecuación de regresión logística para hacer una predicción tomando el valor aproximado por este para las características del *dataset*:

$$h(x) = w^T * x + b$$

$$s(x) = \frac{1}{1 + e^{-h(x)}}$$

Esta ecuación devuelve un valor entre 0 y 1, lo que permite manejar este valor como una probabilidad, donde los resultados de $s(x)$ sean mayores que a 0.5 serán aproximados a 1 y los valores menores que 0.5 se aproximarán a 0. En el caso de obtener un valor exactamente igual a 0.5, se encuentra el caso de mayor incertidumbre donde existente la misma probabilidad de pertenecer a uno de los dos grupos.

El modelo utilizado puede devolver predicciones erradas, por lo que se buscan evaluar los resultados para cada uno de los elementos del *dataset* y minimizar el error para ambas clases. El error para la clase representada por 1 se reducirá a medida que la aproximación del modelo se acerque a 1; mientras que el error para la clase representada por 0, se reducirá a medida que la aproximación del modelo se acerque a 0.

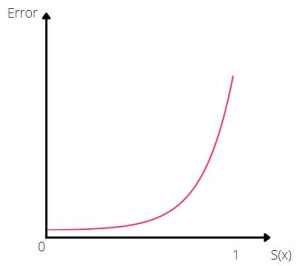

Fig. 1: Visualización del error para $y = 0$

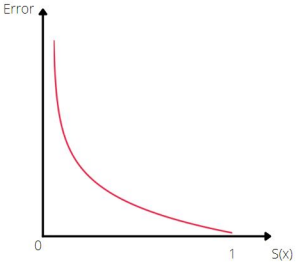

Fig. 2: Visualización del error para $y = 1$

Para medir el error del modelo, se debe multiplicar el logaritmo del valor obtenido por el modelo con el valor real por -1 para medir el error de $y = 1$ y de $y = 0$. Esto va a resultar en que siempre se va a evaluar los resultados dentro de ambos grupos eliminando del modelo la probabilidad de que un mismo valor pertenezca a dos grupos a la vez. Esto lleva a la siguiente función de pérdida:

$$L = \frac{-1}{n} \sum_{i=0}^{n} (y_i \times log(s(x_i)) + (1 - y_i) \times log(1 - s(x_i)))$$

Como se busca minimizar L, se debe derivar respecto a cada w_j:

$$\frac{\partial L}{\partial w_j} = \sum_{i=0}^{n} (y_i - h(x_i)) * (-x_i^j)$$

Posteriormente, se debe multiplicar la derivada de cada w_j por el *learning rate* (α) y reducirle esta cantidad para llegar al mínimo absoluto.

B. Support Vector Machine (SVM)

Support Vector Machine es un algoritmo de aprendizaje supervisado, que busca generar un ($w^t * x + b$) que maximice las distancias entre dos categorías o en términos coloquiales

la ”calle” mas ancha que separe la data.

Esto es lo mismo que maximizar la distancia entre dos planos paralelos, tal que cada uno contengan todos los *samples* de esa categoría, arribando al siguiente modelo:

$$w^T * x_i + b \geq 1 \text{ si } y_i = 1$$
$$w^T * x_i + b \leq -1 \text{ si } y_i = -1$$

Esto por conveniencia matemática se puede expresar como:

$$y_i(w^T * x_i + b) \geq 1$$

Dado este modelo la distancia entre los planos externos, es igual a $d = (x_+ - x_-)(w/||w||)$, lo que se puede reducir a $(x_+w - x_-w)/||w||$, y empleando la ecuación anterior, $(1 - b + 1 + b)/||w||$. Resultando en $d = 2/||w||$, que es lo que queremos maximizar, lo que es equivalente a minimizar $||w||/2$ y para facilitar mejor el calculo de las derivadas se elevara al cuadrado w, concluyendo en $d = ||w||^2/2$.

Una vez entendido esto, es importante explicar que existen dos tipo de *svm*, *hard* y *soft*, el primero maximiza la distancia sin permitir errores lo cual lo hace extremadamente sensible a *outliers*, por otra parte la versión *soft* si permite errores, lo cual converge en menor probabilidad de un *overfitting*. Debido a que posteriormente emplearemos la versión *soft* por razones que detallaremos mas adelante, se explicara este modelo.

Ya que permitiremos errores en nuestra predicción el modelo seria de la siguiente manera, $y_i(w^T * x_i + b) \geq 1 - E_i$, siendo la función de error, $E = max(0, 1 - y_i(w^T * x_i + b))$, esta se deriva por el hecho de que por ejemplo, cuando $y_i(w^T * x_i + b) \geq 1$ (la clasificación es correcta), al restarle 1, el valor siempre sera menor o igual a 0, dando un error de 0.

Resultando en,

$$L = \frac{||w||^2}{2} + C * \sum_{i=1}^{n} max(0, 1 - y_i(w^T * x_i + b))$$

Siendo C un hiper-parámetro que permitirá regular la cantidad de error deseado, cuando C tiende al infinito el modelo no admite errores (hard) y al contrario.

Por ultimo ya que queremos minimizar L, derivaremos respeto a w_j y b, obteniendo que para cuando $y_i(w^T * x_i + b) \geq 1$

$$\frac{\partial L(w,b)}{\partial w_j} = w_j \text{ y } \frac{\partial L(w,b)}{\partial b} = 0$$

Caso contrario,

$$\frac{\partial L(w,b)}{\partial w_j} = w_j - C * \sum_{i=1}^{n} y_i * x_i \text{ y } \frac{\partial L(w,b)}{\partial b} = -y_i * C$$

Finalmente una vez obtenidas las derivadas solo es cuestión de dirigirnos en el sentido opuesto a la recta tangente multiplicado por el *learning rate* (α), en un numero de iteraciones previo.

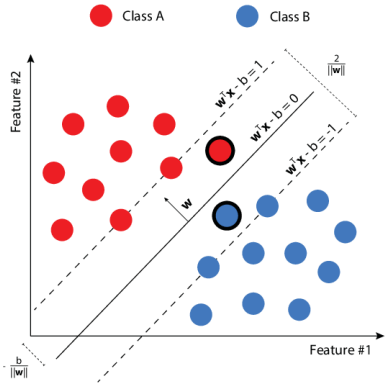

Fig. 3: Visualización

C. *Árboles de decisión*

Un árbol de decisión sirve para separar un conjunto de datos en dos subconjuntos al evaluar una condición y ver donde podría colocarse un nuevo valor ingresado al *dataset*. Tienen la forma de un árbol donde la raíz es el *dataset* original y los pesos de las aristas indican la condición de transición para los demás nodos de decisión que dividen el *dataset* en clases. Para clasificar un nuevo *datapoint* se le asigna una clase a dicho valor. Si hay más clases, se van a hacer más subdivisiones con más nodos de decisión que terminan siendo nodos hojas.

Para entrenar un árbol de decisión, primero se debe determinar el nodo de decisión raíz y la característica mediante las cuales se van a evaluar los datos. Evidentemente, se busca tener un nodo de decisión que separe bien el *dataset*, siendo que las diferentes clases se pueden separar lo más posible. Por lo tanto, la meta es elegir un nodo raíz que nos de la mejor separación de valores. Para esto se pueden utilizar el *gini impurity* y el *gini gain*. Fórmula de *gini impurity*:

$$G = \sum_{i=1}^{c} p(i) * (1 - p(i))$$

Donde c representa las clases del conjunto de datos y $p(i)$ la probabilidad de tomar un punto de la clase i.

Para elegir el nodo raíz, se va probando cada separación posible y la separación que tenga mayor *gini gain* se elegirá. Para probar cada separación, se evalúa cada característica de los datos y se toman todos los “valores únicos” para producir diferentes separaciones. Luego, se calcula el *gini gain* para las separaciones generadas y se toma la separación que genere el mayor *gini gain* como el nodo de decisión raíz.

Con el nodo de decisión raíz, se fija la característica y condición de transición que va a usarse para transicionar hacia los demás nodos de decisión que contienen las particiones

hechas. Arbitrariamente se elige uno de los nodos de decisión creados, para construir el segundo nodo de decisión haciendo exactamente el mismo proceso. Se toma una característica, se evalúan las separaciones posibles para esa característica, se evalúa el gini gain y se tomar el mayor generando la separación. Se entrena el árbol de decisión hasta que los nodos de decisión definan bien todas las clases del dataset.

D. K-Nearest Neighbours

K-Nearest Neighbours o KNN es un algoritmo supervisado que busca encontrar los K vecinos más cercanos a un punto tomado de un *dataset*. KNN se puede aplicar tanto como a la clasificación como a la regresión, en este caso, solo se aplicará a la clasificación.

Durante la etapa de entrenamiento, simplemente se almacenan los datos del *dataset* dentro de una estructura de datos que permitirá realizar la búsqueda de los K vecinos. Durante la etapa de predicción, simplemente se buscan los K vecinos más cercanos en base a la distancia del punto elegido, se obtienen sus etiquetas y se realiza un *voting* para determinar que etiqueta es la que más aparece. Existen dos formas de *votes*, el primero es *majority voting* el cuál encuentra la etiqueta que se repite más veces de los vecinos con 2 etiquetas diferentes como máximo y el segundo es *plurality voting* el que tenga más frecuencia de los vecinos encontrados.

En otras palabras, se tiene una función $f(x) = y$ que va a asignar una etiqueta $y \in 1, 2, 3, \ldots, t$ a un elemento de entrenamiento. Luego se identifican los K vecinos más cercanos para cada punto x_q. Con esto, buscamos determinar que etiqueta permite maximizar la cantidad de elementos que son iguales, en otras palabras, determinaremos que etiqueta es más frecuente. Simplemente se encuentra la moda de las etiquetas de los K vecinos más cercanos. Esto se expresa de la siguiente forma:

$$\delta(y, f(x_i)) = \{1 \text{ si a=b}, 0 \text{ si a}\neq\text{b}\}$$

$$h(x_q) = \underset{y\in\{1,2,3,\ldots,t\}}{argmax} \sum_{i=1}^{k} \delta(y, f(x_i))$$

$$h(x_q) = mode(f(x_1), f(x_2), \ldots, f(x_k))$$

E. K-Fold Cross Validation

Para validar los resultados dentro de los modelos, se utiliza el *K Fold Cross Validation*. El *K Fold Cross Validation* realiza K experimentos donde se toma una parte del *dataset* que representa $\frac{1}{K}$ del *dataset* total como *validation* y el resto del *fold* se usará para *training*. A medida que se realizan los experimentos se ira tomando el siguiente $\frac{1}{K}$ del *dataset* total garantizando que se usarán todos los *folds* como máximo 1 vez para *validation*. El cálculo de *accuraccy* para el metodo de *K Fold Cross Validation* es:

$$A = \frac{1}{K} \sum_{i=1}^{K} A_i$$

Donde A_i representa el *accuraccy* del experimento.

III. Experimentación

A. Proceso general

1) Conversión de valores de la variable dependiente: Lo primero que se realizó fue la conversión de la variable dependiente a 1-0 para la Regresión Logística, mientras que para arboles de decisión, *SVM* y *KNN*, -1 y 1.

2) Normalización de la data: Debido a la naturaleza del *dataset* que se está usando para la experimentación, se normalizaron los datos para mejorar la eficiencia del modelo y evitar posibles errores durante el proceso de aprendizaje de cada uno de los modelos utilizados. Esto es vital para el correcto funcionamiento del svm ya que al basarse en las distancias de los puntos, sin una correcta normalización algunas características pueden influir mas que otras, por lo que se normalizaron cada una de las características del *dataset* de la siguiente forma

$$x = \frac{x - min(x)}{max(x) - min(x)}$$

También cabe destacar que para el caso de los arboles de decisión no es necesario normalizar la data debido a que el algoritmo trata cada característica de forma independiente.

3) Creación de train, testing y validation: Para cada uno de los experimentos, a excepción del *KNN*, se separó el dataset en 70% para *training*, 20% para *validation* y 10% para *testing* tomados de manera aleatoria con la función *train_test_split* de la librería *sklearn*. En el caso del *KNN*, se explicará en su respectivo apartado.

B. Regresión logística

Para cada uno de los experimentos, se emplearon 1000 épocas y se modificó el valor del *learning rate* (α). Lo primero que se realizó en cada experimento, posterior a lo explicado en el apartado de "Proceso general" fue entrenar el modelo para cada elemento del *dataset*. En este paso, en cada época se calculaba las derivadas para ajustar los parámetros w_j del hiper-plano que separa las dos clases de "*Male*" y "*Female*" y reducir el error con la multiplicación de las derivadas y el *learning rate*. Después de modificar estos parámetros, se calculaba el error de *validation* y *testing* y se agregaba estos elementos a una lista para graficarlas respecto a la época en la que se calculó el error. Las siguientes gráficas obtenidas que se muestran a continuación, representan el error del modelo utilizando el conjunto de datos para validación y entrenamiento para los diferentes valores de alfa probados.

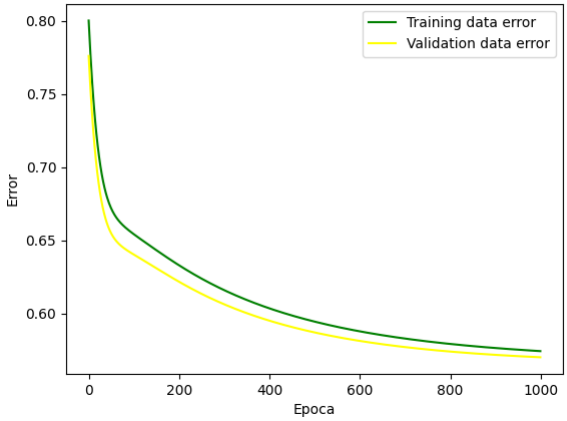

Fig. 4: Gráfico para $alfa = 0.01$

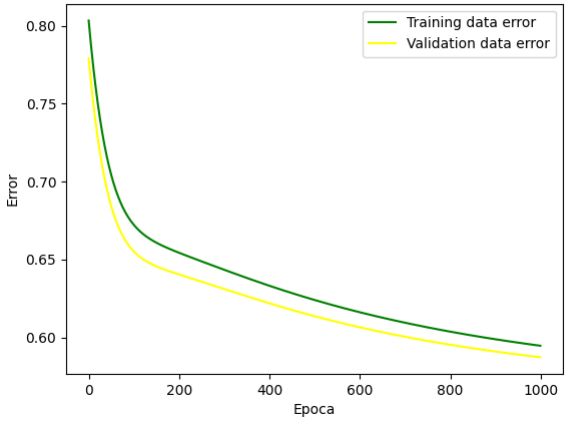

Fig. 5: Gráfico para $alfa = 0.005$

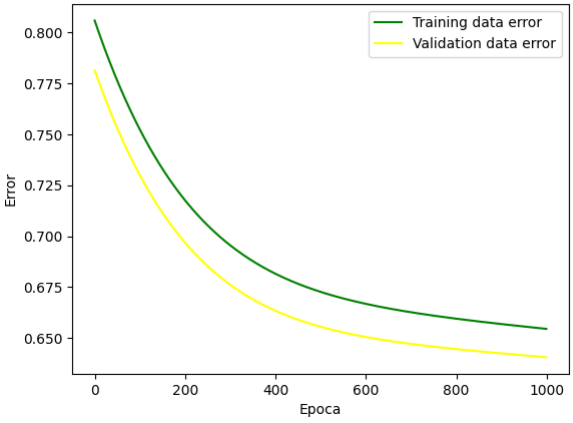

Fig. 6: Gráfico para $alfa = 0.001$

En las gráficas, se puede ver que para valores de alfa que se acerquen más a 1, el algoritmo es capaz de aprender más rápido y reducir el error de aprendizaje del algoritmo. Asimismo, se debe destacar que no hay *overfitting* ni *underfitting* visible para las épocas establecidas. Esto se evidencia en el comportamiento similar de las curvas de error para el *dataset* de *training* y *validation* donde se reduce el error del modelo para los ambos *dataset* y se evidencia el aprendizaje del modelo con los datos del *dataset* de *training*.

Para los datos de *testing*, se obtuvieron los siguientes errores para cada uno de los valores del *learning rate* después de haber concluido el entrenamiento del modelo:

TABLE I: Tabla de errores para cada alfa probado con los datos de *testing*

Valor de alfa	Error
0.01	0.57166827166494
0.005	0.59217846020998
0.001	0.65346782002781

Estos resultados demuestran que el error de *testing* aumenta para valores más bajos de alfa (α), al igual que para los datos de *training* y *validation*.

C. Support Vector Machine (SVM)

Para el caso del SVM, en todas las pruebas el numero de epochs y el learning rate fue constante, definido en 1000 y 0.001 respectivamente, después procedimos a modificar los valores de c, empezamos con un valor bastante alto (1000), con el objetivo de aproximarnos a un hard margin, para lo que obtuvimos valores bastante malos y ruidosos.

Fig. 7: Épocas vs error para c=1000

Obteniendo una accuracy de 0.522, 0.53 y 0.48 para el traning, testing y validation set, respectivamente, al ser demasiado baja la precision con c=1000 resultando el modelo en un underfitting.

Ahora probaremos con c=100, obteniendo lo siguiente.

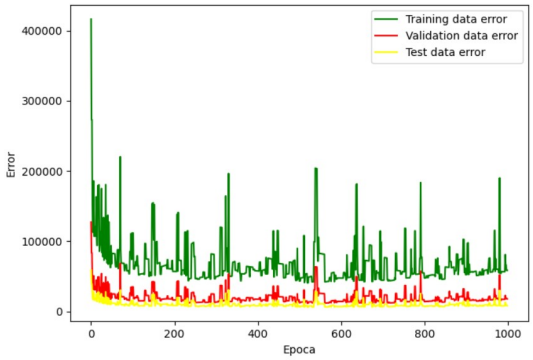

Fig. 8: Épocas vs error para c=100

Obteniendo una accuracy de 0.95, 0.96 y 0.94, lo que es muchisimo mejor.

Debido a la mejora del modelo anterior, entrenamos un modelo con c=10, convergiendo en.

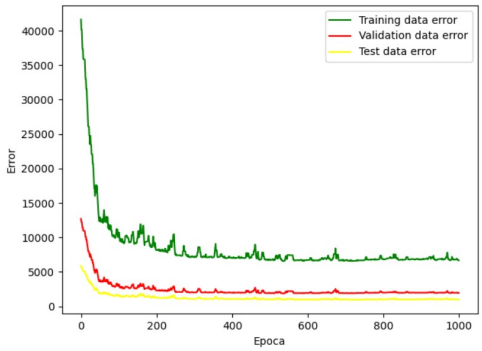

Fig. 9: Épocas vs error para c=10

Obteniendo una accuracy de 0.96, 0.96 y 0.96, lo que supone una mejora del modelo anterior.

Por ultimo realizaremos una prueba con c=1.

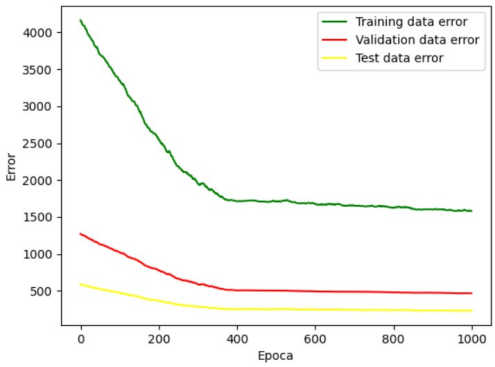

Fig. 10: Épocas vs error para c=1

Obteniendo una accuracy de 0.93, 0.948 y 0.94, lo que supone una disminución respecto al modelo anterior, por lo que concluimos que el modelo mas óptimo es cuando c es igual a 10.

Cabe destacar que la razón por la cual los errores de validación y test están debajo que los de training, se puede deber a distintos factores, como que la última data es muchísimo mas grande que las demás por lo que es más probable que contengan menor errores, y al ser una sumatoria la funcion de error, es logico que las graficas se comporten de ese modo.

D. Árboles de decisión

Para realizar las pruebas de este algoritmo y debido a su naturaleza empleamos 10 fold cross validation, utilizando la métrica de gini impurity. Los resultados son los siguientes, obtuvimos un accuracy promedio de 0.951, lo que es bastante bueno, logrando superar a la regresión logística y al KNN, sin embargo svm es superior.

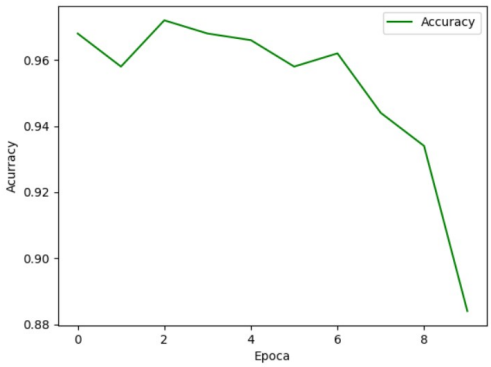

Fig. 11: Épocas vs error

E. K-Nearest Neighbours

Para el KNN, se modificó el proceso general utilizado para los demás en la parte de creación de *train*, *testing* y *validation*.

Al momento de separar el *dataset*, solamente se separa en *training* y *testing* con el método de *K-Fold Cross Validation* explicado en el la sección de Explicación del modelo empleado en el apartado de *K-Nearest Neighbours*. Cabe destacar que para *K-Fold Cross Validation* $k = 10$, y se ingresan la cantidad de vecinos para la experimentación como hiper-parámetro. Para cada uno de los *folds* se adaptó el modelo y se calculó el error para los datos de *training* y *testing*. Posterior al cálculo de errores, se determinó el error total para el modelo y se graficaron los errores para cada uno de los experimentos.

Se realizaron pruebas para 20 vecinos, 40 vecinos y 60 vecinos. A continuación se muestran las gráficas para cada uno:

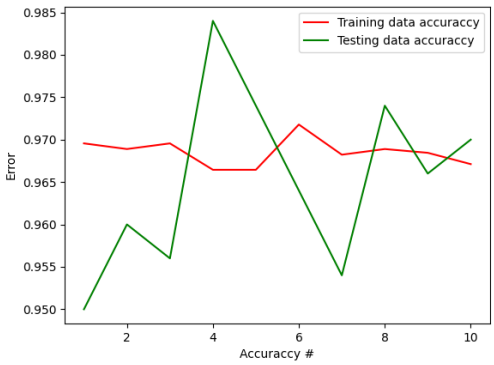

Fig. 12: Gráfico para 20 vecinos

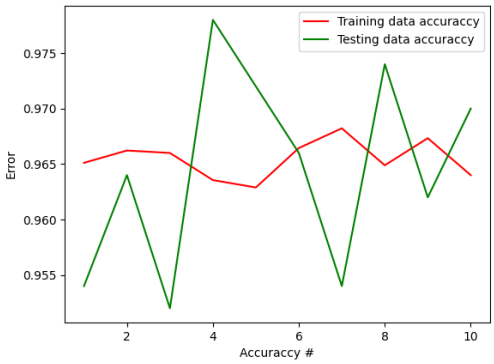

Fig. 13: Gráfico para 40 vecinos

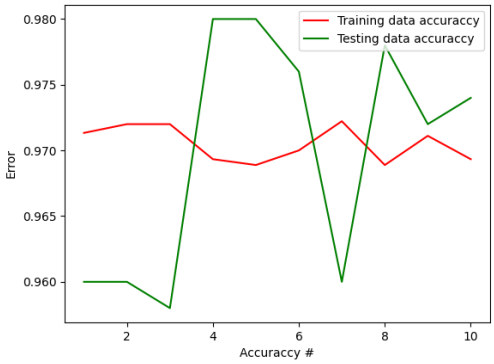

Fig. 14: Gráfico para 60 vecinos

En este gráfico, se muestra que el error para cada experimento no es consistente pero que, por lo general, el error de *training* es más consistente que el error de *testing*. Esto pasa principalmente porque el modelo está adaptado para los datos de *training* por lo que es difícil replicar esos resultados con el conjunto de datos de *testing*, en otras palabras existe un *overfitting* con el modelo.

Para una mejor interpretación del error, se calculó el error total mostrado en la siguiente tabla.

Los resultados del error para mostrados en la tabla muestran, que el error se va reduciendo a medida de que hay mayor cantidad de vecinos. Esto se puede deber a que el modelo es capaz de aprender con mayor capacidad porque en su mayoría usa los datos de *training* al momento de predecir el error para cada uno de los experimentos.

TABLE II: Tabla de *acurracy* total para cada experimento

Vecinos	Accuracy
20	0.4841
40	0.241345
60	0.16174

IV. Conclusiones

En conclusión, al desarrollar los diferentes modelos se ha determinado que, en base a los resultados de la experimentación, el mejor algoritmo clasificador para esta base de datos es *Support Vector Machine*, consideramos esto debido a su flexibilidad al momento de adaptarse a los datos.