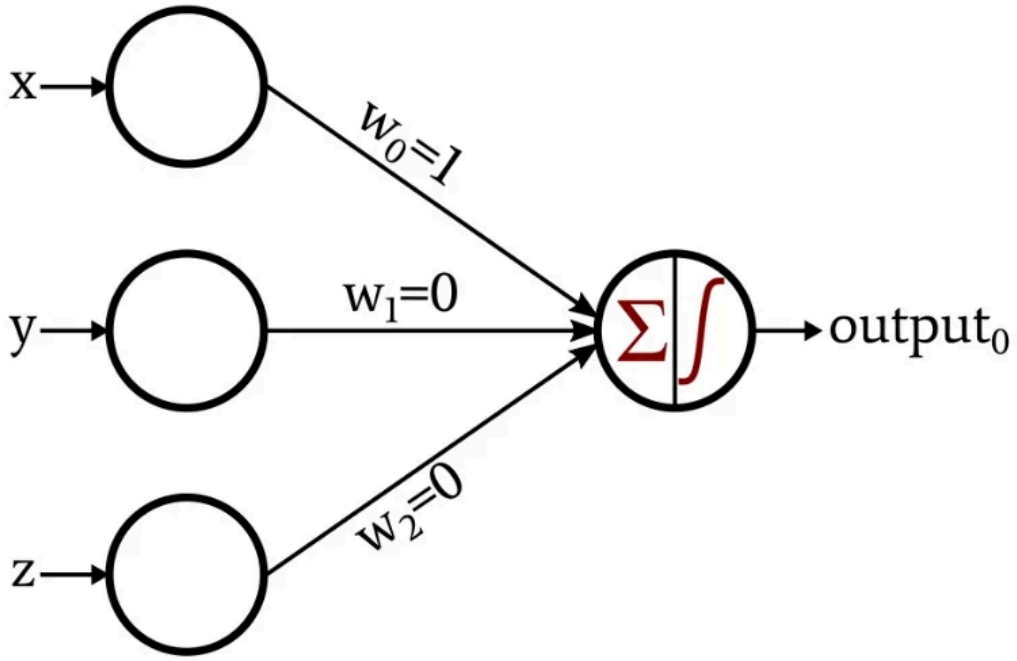
MANUAL TÉCNICO
ALGORITMO GENÉTICO PARA HORARIOS



Introducción

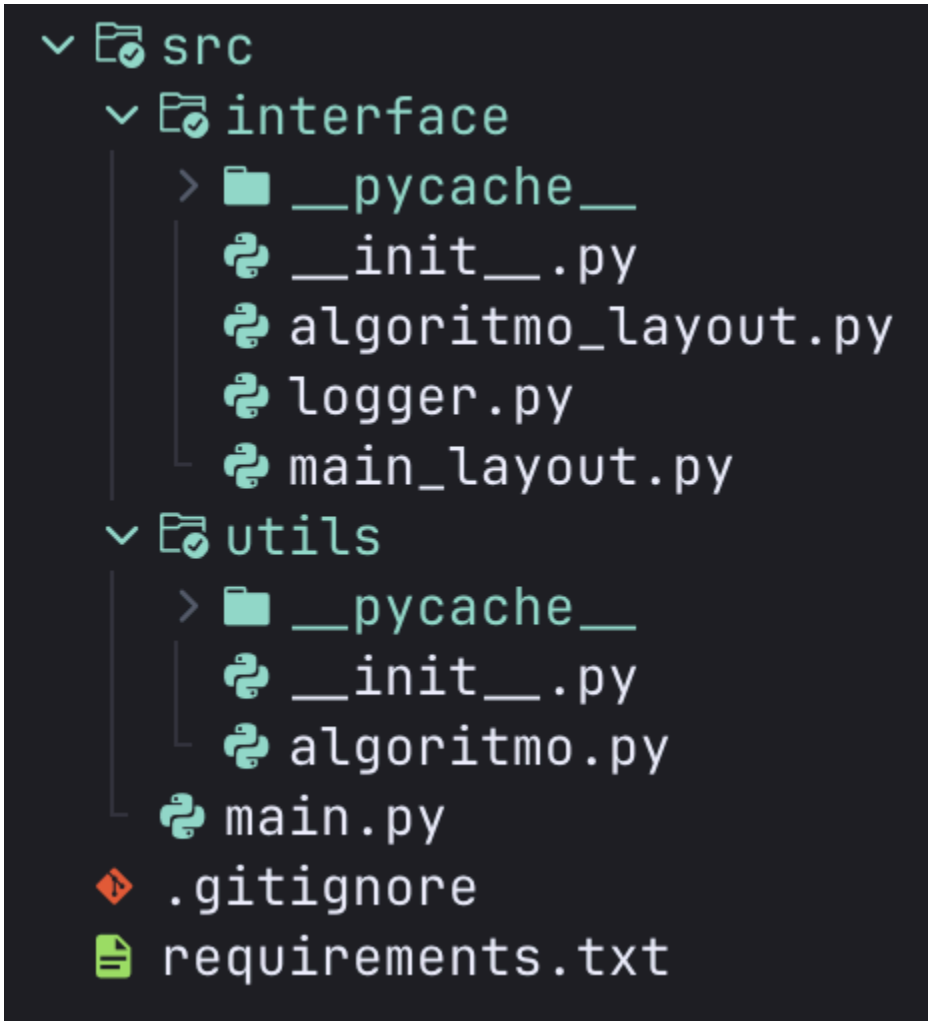
En el presente documento se presenta la documentación técnica del perceptrón simple para detección de cáncer.

Arquitectura del Proyecto

El proyecto está completamente desarrollado en Python usando las siguientes librerías:

- PyQT5: Para el desarrollo de la interfaz de la aplicación
- Matplotlib: Para la generación y visualización de gráficas
- Scikit-Learn: Para la obtención del dataset de Breast Cancer Wisconsin

Como se puede asumir por el uso de la librería PyQT5 la aplicación es una de escritorio que conecta con el perceptrón de forma acoplada.



El proyecto está dividido en 2 carpetas principales.

interface

Es la carpeta que contiene todo lo relacionado a la interfaz de usuario con PyQT5, además de visualizadores para graficas.

utils

Contiene los archivos con lo necesario para el funcionamiento del perceptrón.

Perceptron

Ambiente del Algoritmo

El ambiente del algoritmo es una clase que contiene todos los métodos necesarios para el funcionamiento del mismo incluyendo pesos, características, data y columnas.

```python
class AmbienteAlgoritmo:
    def preparar_data(self):
        data = load_breast_cancer()
        self.x = data.data #type: ignore
        self.y = data.target #type: ignore
        self.feature_names = list(data.feature_names) #type: ignore
```

Preparación de la información

El ambiente del algoritmo contiene un método que permite la carga de toda la información al inicio para su posterior uso a partir del método load_breast_cancer de scikit-learn.

```python
class AmbienteAlgoritmo:
    def preparar_data(self):
        data = load_breast_cancer()
        self.x = data.data #type: ignore
        self.y = data.target #type: ignore
        self.feature_names = list(data.feature_names) #type: ignore
```

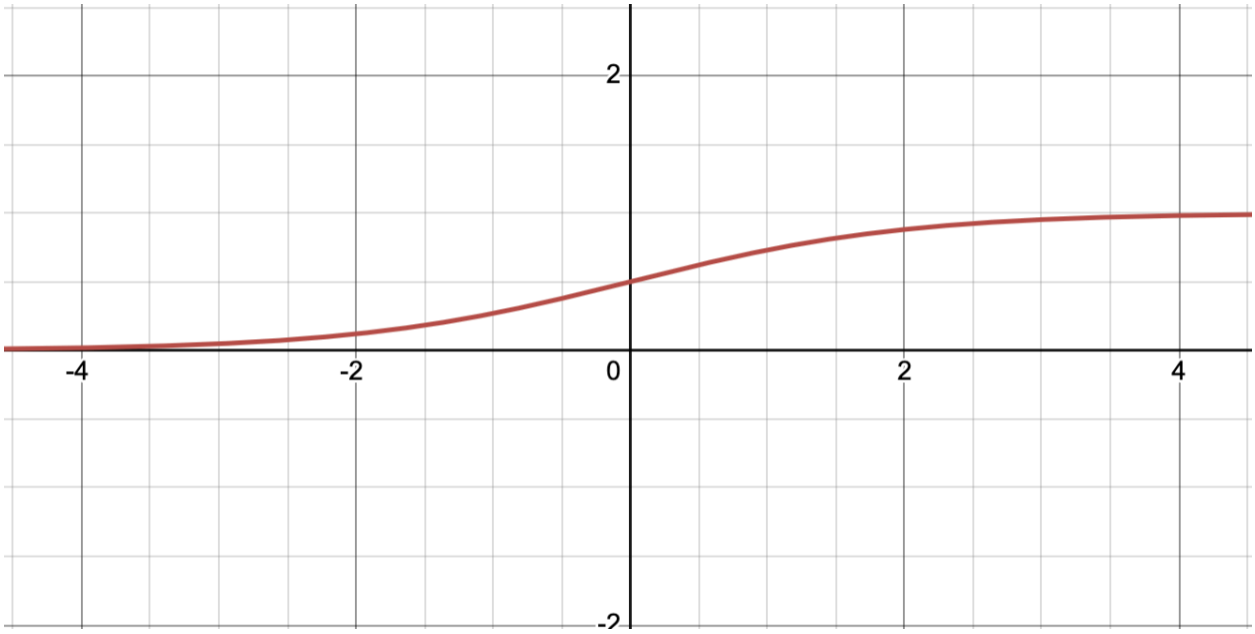
Función Sigmoide (Función de Activación)

Para la función de activación del perceptrón se utilizó la función sigmoide.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

El papel de la función de activación es decidir cuándo activar la neurona y convertir un conjunto de datos en una decisión binaria, en este caso cuando el tumor es benigno (0) o cuando es maligno (1).

Se escogió la función sigmoide como función de activación por dos razones: primero, es una función no lineal, lo que significa que el que la neurona se active o no no depende de una proporción lineal, sino de un crecimiento natural, la segunda es que es una función suave, lo que significa que no tiene escalones ni cambios bruscos, como se mencionó anteriormente el crecimiento es natural por lo que los cambios son suaves.



Otra razón es que esta es una función derivable respecto a una función escalonada, y esto nos permite saber el cambio que tendrá la salida respecto a cada entrada.

$$\sigma(z) = \sigma(z) * (1 - \sigma(z))$$

Esta es una función probada en el mundo real como una función que permite un entrenamiento de neuronas y un cambio del gradiente óptimo.

La implementación a nivel de código de la función sigmoide es la siguiente:

```
def sigmoide(self, z):
    return 1 / (1 + np.exp(-z))
```

Normalización

Se usó la normalización para escalar los datos ingresados para que sigan la misma escala.

Que los diferentes conjuntos de datos estén en la misma escala es importante para que durante el entrenamiento no beneficie a ciertas características sobre otras, ya que si hubiese características que tuviesen valores en los miles y otra en las unidades, el perceptrón beneficiaria a la primera por tener valores más grandes, por lo que al normalizar el entrenamiento le da igual importancia a todas las características.

En este caso la normalización se hace por medio de la fórmula de normalización Z-score

$$X = \frac{X - \mu}{\sigma}$$

Esta fórmula resta el conjunto de datos menos la media de estos y los divide entre su desviación estándar dando como resultado un conjunto de datos cuya media es 0 y su desviación estándar es 1, a través de esta fórmula es posible normalizar los datos para que todos funcionen bajo la misma escala.

Su implementación en el código es la siguiente.

```
def normalizar(self, X):
    # Normalización: Z = (X - media) / desviación
    media = np.mean(X, axis=0)
    desviacion = np.std(X, axis=0)
    return (X - media) / desviacion
```

Se usa la librería numpy para la obtención de la media y la desviación estándar de los datos.

División de Datos

Debido a que es recomendado no usar el 100% de los datos para realizar el entrenamiento ya que no habrian datos para realizar las pruebas, es necesario dividir los datos en un

porcentaje para que cierta parte sean los datos de entrenamiento y otra parte los datos de prueba, para esto se usa numpy del siguiente modo.

```python
def dividir_datos(self, X, y, porcentaje_entrenamiento=0.8):
    # Mezclar aleatoriamente
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)

    X = X[indices]
    y = y[indices]

    corte = int(porcentaje_entrenamiento * X.shape[0])
    X_train, X_test = X[:corte], X[corte:]
    y_train, y_test = y[:corte], y[corte:]

    return X_train, X_test, y_train, y_test
```

Primero todos los datos son mezclados aleatoriamente para no siempre obtener los mismos, luego en base a un porcentaje ingresado se 'corta' la matriz de datos y se obtiene de ahí los datos de entrenamiento y los de prueba.

Entrenamiento

El entrenamiento del perceptrón es la parte más importante del algoritmo, esta se realiza en diferentes partes, y está implementada del siguiente modo.

```python
def entrenar(self, idx_x, idx_y, epochs, eta, porcentaje_entrenamiento, callback_actualizar_plot = None):
    # Seleccionar características
    X_selected = self.x[:, [idx_x, idx_y]]
    y = self.y

    # Normalizar
    X_selected = self.normalizar(X_selected)

    # Dividir
    X_train, X_test, y_train, y_test = self.dividir_datos(X_selected, y, porcentaje_entrenamiento)

    # Se inicializan los pesos (2 características y 1 bias)
    self.pesos = np.random.rand(3)
    self.errors = []

    # Agregar columna de unos (bias)
    X_train_bias = np.c_[np.ones((X_train.shape[0], 1)), X_train]

    # Se recorren las epocas
    for epoch in range(epochs):
        total_error = 0
        for x_i, target in zip(X_train_bias, y_train):
            z = np.dot(x_i, self.pesos)
            output = self.sigmoide(z)
            error = target - output
            self.pesos += eta * error * x_i
            total_error += error ** 2

        self.errors.append(total_error)
        Logger.instance().log(f"Epoca {epoch+1}, Error: {total_error:.4f}")
        time.sleep(0.1)

        if callback_actualizar_plot:
            callback_actualizar_plot(self.pesos, X_selected, y, idx_x, idx_y)

    # Evaluar exactitud en pruebas
    X_test_bias = np.c_[np.ones((X_test.shape[0], 1)), X_test]
    accuracy = self.accuracy(X_test_bias, y_test)
    Logger.instance().log(f"Exactitud final: {accuracy * 100:.2f}%")

    return {
        "pesos": self.pesos,
        "errores": self.errors,
        "precision": accuracy
    }
```

La primera parte es la selección de las características, la normalización de la información y su división usando los métodos explicados anteriormente.

```python
# Seleccionar características
X_selected = self.x[:, [idx_x, idx_y]]
y = self.y

# Normalizar
X_selected = self.normalizar(X_selected)

# Dividir
X_train, X_test, y_train, y_test = self.dividir_datos(X_selected, y, porcentaje_entrenamiento)
```

Luego se hace la inicialización de los pesos y el sesgo (y lista de error).

```
# Se inicializan los pesos (2 características y 1 bias)
self.pesos = np.random.rand(3)
self.errors = []
```

Se crea simplemente un arreglo con 3 valores donde los dos primeros son los pesos para las dos características evaluadas y el tercero es un valor que servirá de sesgo. Además de esto se inicializa un arreglo vacío para almacenar el error obtenido por época.

Luego se agrega el bias a los datos de entrenamiento para incluir el sesgo en el entrenamiento.

```
# Agregar columna de unos (bias)
X_train_bias = np.c_[np.ones((X_train.shape[0], 1)), X_train]
```

Luego se realiza la fase de entrenamiento.

```
# Se recorren las epocas
for epoch in range(epochs):
    total_error = 0
    for x_i, target in zip(X_train_bias, y_train):
        z = np.dot(x_i, self.pesos)
        output = self.sigmoide(z)
        error = target - output
        self.pesos += eta * error * x_i
        total_error += error ** 2
```

Se realiza el entrenamiento a través de las épocas ingresadas.

- Ciclo Interno

Se realiza un ciclo interno para recorrer todas las muestras de los datos de entrenamiento.

```
for x_i, target in zip(X_train_bias, y_train):
```

- Cálculo de Potencial de Activación

Para clasificar cada muestra es necesario calcular su potencial de activación para saber 'cuánto' activa la neurona esa muestra específica.

Para esto se hace una suma ponderada, y se multiplican los valores de esa muestra por los pesos y luego se le agrega el sesgo del siguiente modo.

$$z = w_0 + w_1 x_1 + w_2 x_2$$

Esta multiplicación entre los pesos y los valores puede ser interpretado como el producto punto de ambos ya que los dos son matrices de 1x3. Por lo que matemáticamente esta operación también puede ser.

$$z = \overline{x_i} \cdot \overline{w}$$

Dando como resultado un único valor.

Esta implementación está hecha del siguiente modo.

```
z = np.dot(x_i, self.pesos)
```

Se usa el método dot de numpy para calcular el producto punto entre ambos arreglos.

- Aplicación de la Función de Activación

  Luego de obtener el Potencial de Activación de la muestra se le aplica la Función de Activación (Sigmoide explicada anteriormente) para saber si la muestra activa o no la neurona.

```
output = self.sigmoide(z)
```

- Cálculo del Error

  Luego de obtener la activación del perceptrón se calcula el error de activación, debido a que la muestra está etiquetada con su resultado es posible ajustar los pesos y el sesgo para que las muestras cada vez cumplan con la etiqueta, el error es una resta simple.

$$error = y_i - \hat{y}$$

  Donde y_i es el valor esperado para la muestra actual e $\hat{y}$ es el valor obtenido por la función de activación, por lo que mientras más se aleje de 0 peor es el resultado, si se esperaba 0 y se obtuvo 1, tendrá error de -1, si se esperaba 1 y se obtuvo 0 se tendrá error de 1, en ambos casos hay un error.

  La implementación es simple del siguiente modo.

```
error = target - output
```

- Aprendizaje (Ajuste de Pesos)

  Luego de saber que tan equivocado estaba el resultado obtenido de el resultado esperado se hace el ajuste de pesos, para este ajuste se hace sumando a los pesos actuales la multiplicación de la muestra actual por el error obtenido por la tasa de aprendizaje (η) matemáticamente:

$$\overline{w} = \overline{w} + \eta \cdot error \cdot \overline{x_i}$$

Gracias a este ajuste de los pesos se espera corregir la clasificación realizada para que la frontera de decisión lo haga mejor. Es posible ajustar η para saber que tan grande es el cambio que se realiza, mientras más grande sea η más grandes serán los cambios en los pesos.

La implementación es simple de este modo.

```
self.pesos += eta * error * x_i
```

- Cálculo del error total

Se hace el cálculo del error total sumando el error actual al cuadrado al error total, se hace al cuadrado ya que esto evita que los números negativos afecten la muestra y también permite que los valores con más error sean más penalizados y se muestre así. Su implementación es simple.

```
total_error += error ** 2
```

Cálculo de la exactitud (accuracy)

Después de realizar el proceso de entrenamiento es necesario calcular la exactitud final del perceptrón, para esto se usan los datos de prueba agregándoles la columna bias para que el arreglo tenga la misma forma que el de prueba.

```
X_test_bias = np.c_[np.ones((X_test.shape[0], 1)), X_test]
accuracy = self.accuracy(X_test_bias, y_test)
```

Para el cálculo de la exactitud es simplemente hacer predicciones con los valores de prueba sobre el perceptrón para calcular cuántos de ellos cumplen con lo que la función sigmoide debería de calcular sacando la media de los mismos.

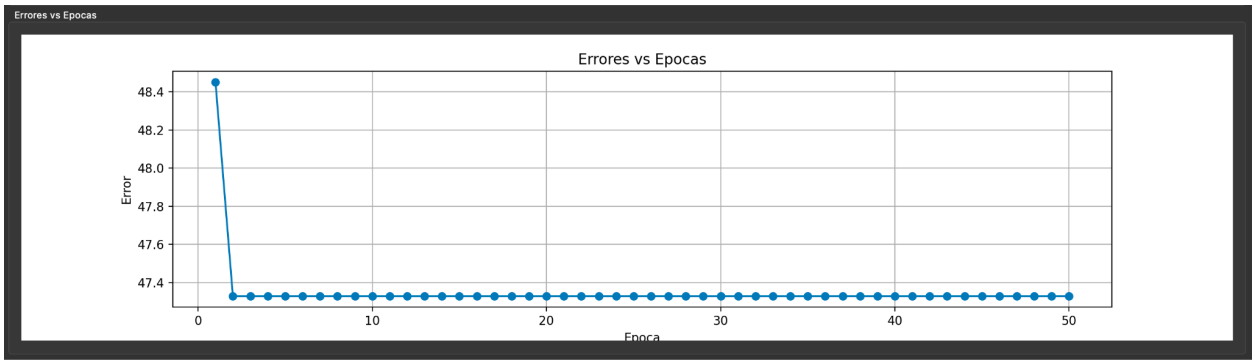
```python
def predecir(self, X_bias):
    z = np.dot(X_bias, self.pesos)
    return np.where(self.sigmoide(z) >= 0.5, 1, 0)

def accuracy(self, X_bias, y):
    y_pred = self.predecir(X_bias)
    return np.mean(y_pred == y)
```

Notas Finales del Algoritmo

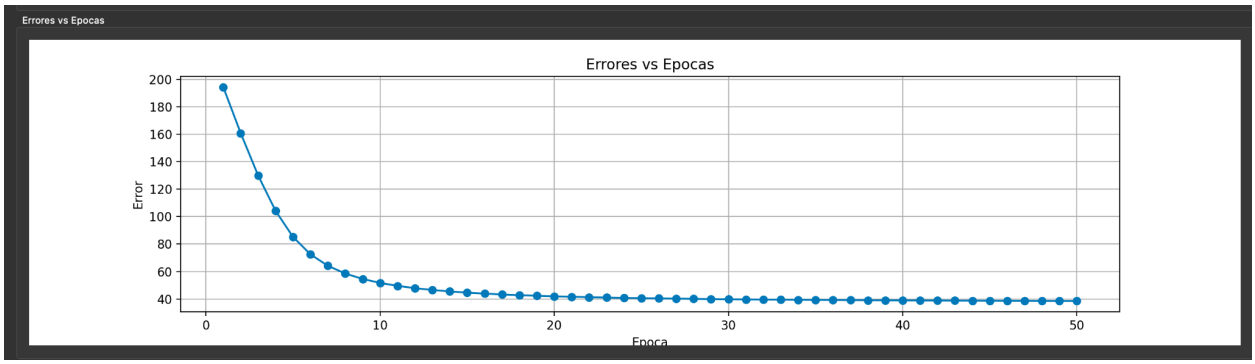
Uso de η

El algoritmo hace uso de eta para la tasa de aprendizaje, esta tasa indica en qué cantidad serán modificados los pesos cada vez que estos sean ajustados.

Para una tasa de aprendizaje alta ($\eta = 1$) el perceptrón aprende en la primera generación pero se estanca al poco tiempo ya que los cambios son tan grandes que no encuentra mínimos para aumentar su fineza.



Para tasas de aprendizaje pequenas ($\eta = 0.001$) el perceptrón se ajusta lentamente hacia una frontera con pocos errores, aunque llega a un mínimo el ajuste es lento.



Si se usa una tasa de aprendizaje intermedia ($\eta = 0.01$) el perceptrón se ajusta velozmente y encuentra un buen resultado en poquisimas generaciones.