

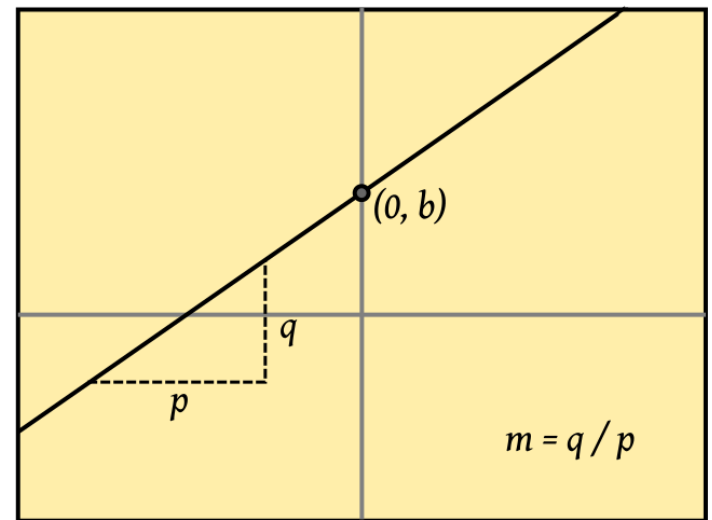
Clasificadores lineales y scikit-learn

Joaquín Pineda Gutiérrez
Luis Garrido Morillo

Clasificadores lineales

- ▶ Clasifican basándose en la combinación lineal de los atributos mas un termino independiente.
- ▶ Sus atributos deben ser numéricos.
- ▶ La importancia de los atributos se mide con los pesos.
- ▶ Son clasificadores binarios

Función lineal



$$y = mx + b$$

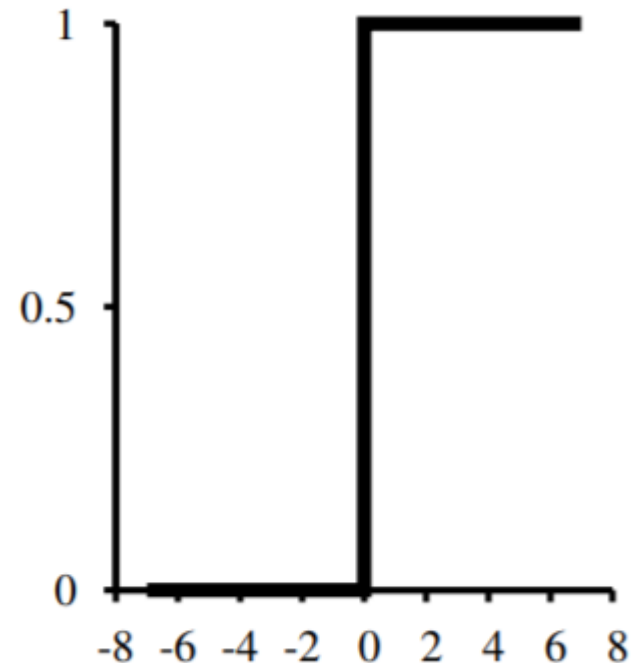
Clasificadores lineales

- ▶ Para la primer parte del trabajo se han implementado 2 tipos de clasificadores lineales:
 - PERCEPTRON
 - Versión estocástica con función umbral
 - REGRESIÓN LOGISTICA
 - Minimización del error cuadrático
 - Versión estocástica con función sigma
 - Versión batch con función sigmoide
 - Maximización de la verosimilitud
 - Versión estocástica con función sigma
 - Versión batch con función sigmoide

PERCEPTRON

Función Umbral

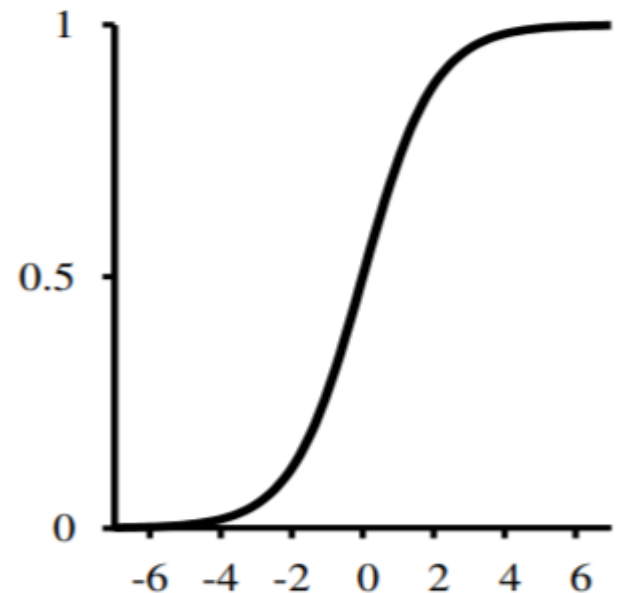
- ▶ Función usada para decidir en qué clase estará el elemento que queremos clasificar.
- ▶ Si el valor calculado por el perceptrón supera un umbral prefijado se considera valor 1 y en otro caso valor 0.



REGRESIÓN LOGISTICA

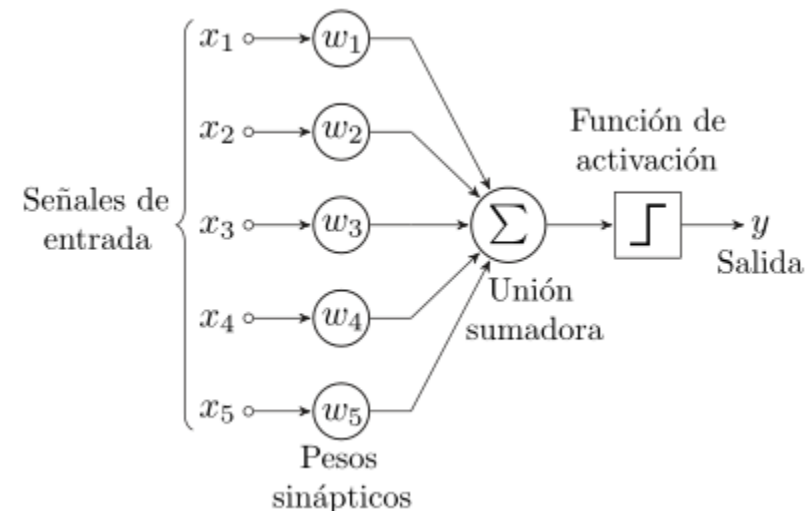
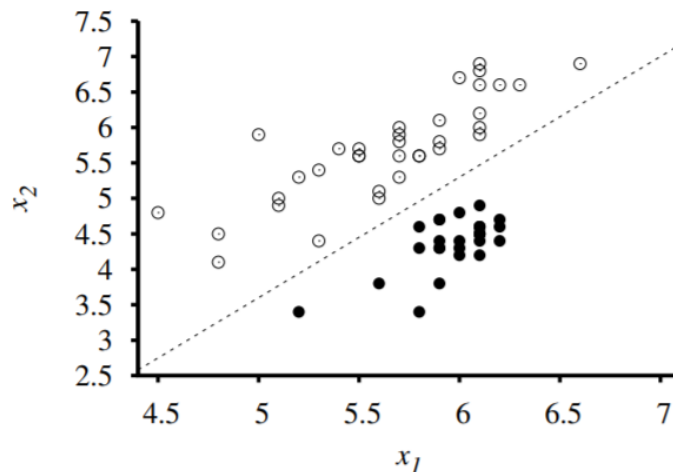
Función Sigmoide

- ▶ Función para decidir en qué clase estará el elemento que queremos clasificar.
- ▶ Da un grado de seguridad entre 0 y 1. Cuanto mas cerca de 1 más seguro estamos de que pertenece a una clase y cuanto más cerca de 0 a la otra.
- ▶ El valor intermedio 0,5 indica inseguridad total, no sabemos como clasificarlo.



PERCEPTRON

- ▶ Se basa imitando el funcionamiento de una única neurona.
- ▶ Establece una frontera que separa el conjunto de datos.
- ▶ No funciona bien en conjunto linealmente no separables.



PERCEPTRON

Estocástico

- ▶ Los pesos se van actualizando en cada iteración de los elementos del conjunto de entrenamiento.
- ▶ “y” es el resultado esperado.
- ▶ “o” es la función umbral

$$w_i \leftarrow w_i + \eta x_i (y - o)$$

$$o = \text{umbral}(\mathbf{w} \cdot \mathbf{x})$$

REGRESIÓN LOGISTICA

Minimización del error cuadrático

- ▶ Se basa en medir cuánto se aleja el valor que devuelve el clasificador de lo que debería para ir ajustando los pesos.
- ▶ Para medir la desviación usamos el error cuadrático el cual deseamos minimizar.

$$L_2(h_{\mathbf{w}}, D) = \sum_j (y^{(j)} - o^{(j)})^2$$

Minimización del error cuadrático Estocástico

- ▶ Los pesos se van actualizando en cada iteración de los elementos del conjunto de entrenamiento.
- ▶ “y” es el resultado esperado.
- ▶ “o” es la función sigma

$$w_i \leftarrow w_i + \eta x_i (y - o) o (1 - o)$$

$$o = \textit{sigma}(\mathbf{w} \cdot \mathbf{x})$$

Minimización del error cuadrático

Batch

- ▶ Los pesos se actualizan una vez por cada epoch, es decir, una única vez después de iterar sobre todo el conjunto de entrenamiento.

$$w_i \leftarrow w_i + \eta \sum_{1 \leq j \leq N} x_i^{(j)} (y^{(j)} - o^{(j)}) o^{(j)} (1 - o^{(j)})$$

REGRESIÓN LOGÍSTICA

Maximización de la verosimilitud

- ▶ Toma un enfoque probabilístico como alternativa a la minimización del error cuadrático.
- ▶ La idea es buscar un peso que dado un ejemplo maximice las probabilidades de que ese ejemplo pertenezca a la clase que realmente tiene asignada.

$$P(c = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}} \quad P(c = 0|\mathbf{x}) = \frac{1}{1 + e^{\mathbf{w} \cdot \mathbf{x}}}$$

REGRESIÓN LOGÍSTICA

Maximización de la verosimilitud

- ▶ Normalmente se suele usar una función equivalente, la log-verosimilitud que es más fácil de calcular.

$$\mathcal{LL}(\mathbf{w}) = - \sum_{\mathbf{x}^{(j)} \in D^+} \log(1 + e^{-\mathbf{w} \cdot \mathbf{x}}) - \sum_{\mathbf{x}^{(j)} \in D^-} \log(1 + e^{\mathbf{w} \cdot \mathbf{x}})$$

Maximización de la verosimilitud Estocástico

- ▶ Los pesos se van actualizando en cada iteración de los elementos del conjunto de entrenamiento.
- ▶ “y” es el resultado esperado.
- ▶ “o” es la función sigma

$$w_i \leftarrow w_i + \eta(y - \sigma(\mathbf{w}\mathbf{x}))x_i$$

$$o = \text{sigma}(\mathbf{w} \cdot \mathbf{x})$$

Maximización de la verosimilitud

Batch

- ▶ Los pesos se actualizan una vez por cada epoch, es decir, una única vez después de iterar sobre todo el conjunto de entrenamiento.

$$w_i \leftarrow w_i + \eta \sum_j [(y^{(j)} - \sigma(\mathbf{w}\mathbf{x}^{(j)}))x_i^{(j)}]$$

Implementación

- Todos los clasificadores siguen el mismo esquema

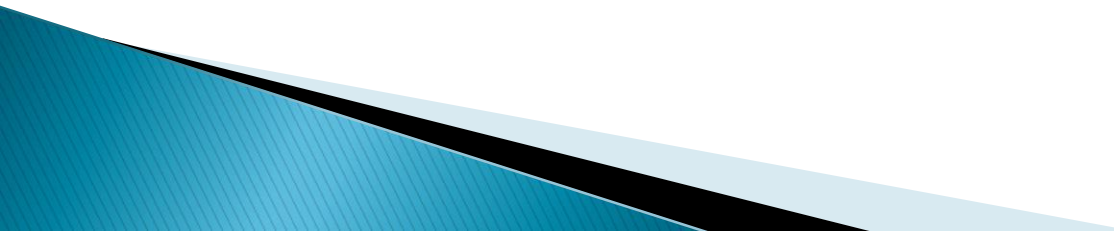
```
class Clasificador():  
  
    def __init__(self, clases, norm=False):  
        None  
  
    def entrena(self, entr, clas_entr, n_epochs, rate=0.1, pesos_iniciales=None, rate_decay=False):  
        None  
  
    def clasifica_prob(self, ej):  
        None  
  
    def clasifica(self, ej):  
        None  
  
    def evalua(self, validacion):  
        None  
  
    def imprime(self):  
        None
```

Implementación

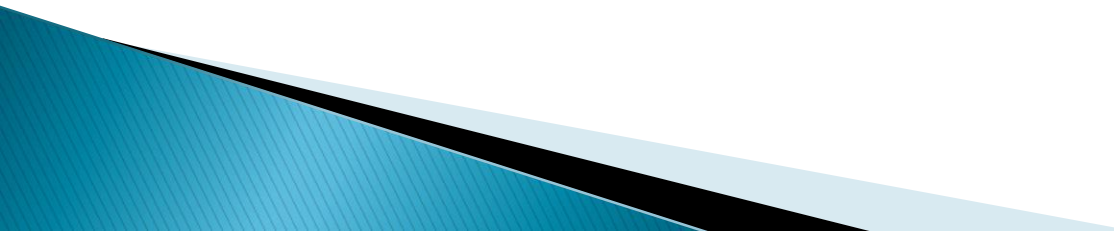
- Todas las funciones de entrenamiento están implementadas igual

```
def entrena(conjunto, resultados, clases, n_epochs, rate_inicial, pesos_iniciales, rate_decay, estocastico=True, summary=None):
    pesos = None
    rate = rate_inicial
    if pesos_iniciales == None:
        pesos = clasificador.genera_pesos(len(conjunto[0]))
    else:
        pesos = pesos_iniciales
    epoch = 0
    n_errors = 1
    while epoch < n_epochs and n_errors != 0:
        n_errors = 0
        for index in range(0, len(conjunto)):
            prediccion = clasificador.calcular_prediccion(conjunto[index], pesos, clases, is_sigma=True)
            if prediccion != resultados[index]:
                n_errors += 1
            if estocastico:
                pesos = ajusta_pesos_estocastico(conjunto[index], pesos, clasificador.busca_resultado(resultados[index], clases), rate)
        if not estocastico:
            pesos = ajusta_pesos_batch(conjunto, pesos, resultados, clases, rate)
        if rate_decay:
            rate = clasificador.decaer_ratio(rate_inicial, epoch)
        if summary != None:
            summary.add_epoch(n_errors/len(conjunto))
            summary.add_magnitud(clasificador.error_verosimilitud(conjunto, pesos))
        epoch += 1
    return pesos
```


Implementación

1. Primero se crean los pesos aleatorios
 2. Luego se comprueba la predicción de cada ejemplo para esos pesos y sumamos los errores para las graficas
 3. Si es estocástico actualizamos los pesos
 4. Al terminar si es batch actualizamos los pesos
 5. Finalmente actualizamos la clase que guarda los datos de la grafica y repetimos hasta alcanzar el numero de epochs
- 

One VS rest

- ▶ One Vs rest es una técnica para poder clasificar entre más de dos clases usando los clasificadores lineales que son binarios.
 - ▶ Consiste en entrenar un clasificador por cada clase que sea capaz de diferenciar entre esa clase y el resto.
 - ▶ Al final se elige la clase que haya devuelto mayor seguridad, es decir, la que de un número más cercano a 1.
- 

One VS rest Implementación

- ▶ Para cada clase entrenamos un clasificador que sepa diferenciar esa clase del resto:

```
def entrena(self, entr, clas_entr, n_epochs, rate=0.1, rate_decay=False):  
    clasificadores = []  
    for clase in self.clases:  
        print("clase: ", clase)  
        clases = ['other', clase]  
        resultados = replace_clases(clase, 'other', clas_entr)  
        clasificador = obtiene_clasificador(self.clasificador, clases, self.estocastico, self.normaliza)  
        clasificador.entrena(entr, resultados, n_epochs, rate=rate, rate_decay=rate_decay)  
        clasificadores.append(clasificador)  
    self.entrenado = True  
    self.clasificadores = clasificadores
```

One VS rest Implementación

- Reemplazamos las otras clases por una clase auxiliar y seleccionamos el algoritmo deseado para entrenar

```
def replace_clases(mantener, otros, original):  
    result = []  
    for elemento in original:  
        if elemento == mantener:  
            result.append(elemento)  
        else:  
            result.append(otros)  
    return result
```

```
def obtiene_clasificador(tipo, clases, estocastico, normalizar):  
    if tipo == 'perceptron':  
        return perceptron.Perceptron(clases, norm=normalizar)  
    if tipo == 'verosimilitud':  
        return maximizar.Maximizar(clases, norm=normalizar, estocastico=estocastico)  
    if tipo == 'regresion':  
        return regresion.Regresion(clases, norm=normalizar, estocastico=estocastico)
```

One VS rest

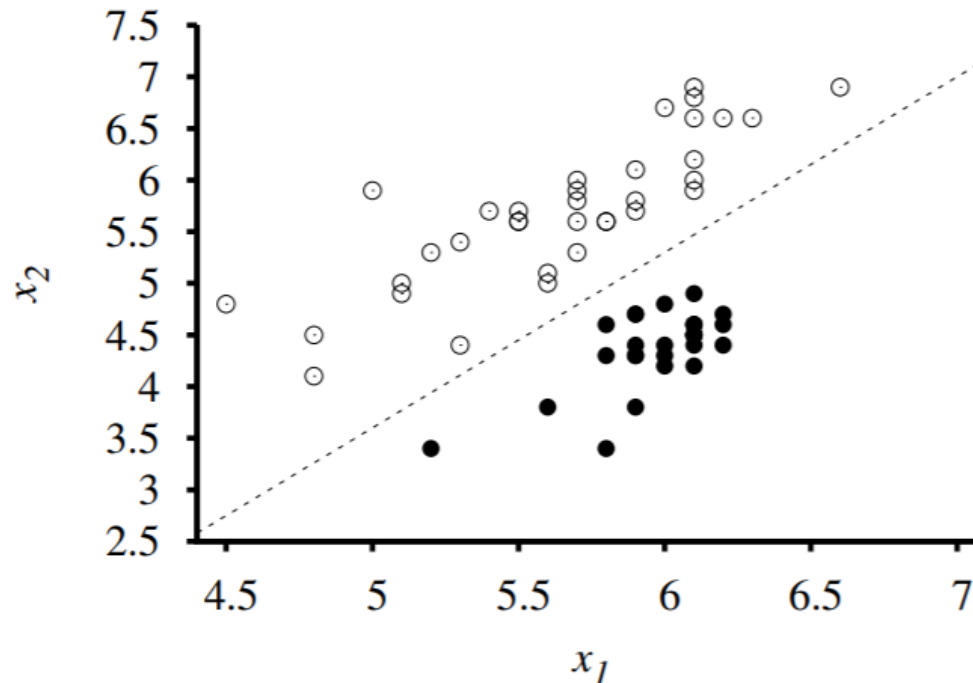
Implementación

- Para clasificar, usamos todos los clasificadores con el ejemplo y elegimos el que proporcione mejor resultado

```
def clasifica_prob(self, ej):  
    mejor = None  
    mejor_i = None  
    for i in range(0, len(self.clasificadores)):  
        clasificador = self.clasificadores[i]  
        res = clasificador.clasifica_prob(ej)  
        if mejor == None or res > mejor:  
            mejor = res  
            mejor_i = i  
    return (self.clases[mejor_i], mejor)
```

Generador de conjuntos

- ▶ Para generar conjuntos elegimos de forma aleatoria una serie de pesos que formaran nuestra frontera



Generador de conjuntos

- Luego es suficiente con generar puntos aleatorios y decidir por medio de alguna función de las estudiadas en qué parte estarían. Nosotros hemos elegido la función de predicción del perceptron.

```
def generar_conjunto_aleatorio(rango, dim, tam, separable=True, clases=None):
    hiperplano = Clasificador.genera_pesos(dim)
    conjunto = []
    soluciones = []
    for i in range(0, tam):
        atributos = []
        for j in range(0, dim):
            atributos.append(random.randint(-rango, rango))
        prediccion = Clasificador.calcular_prediccion(atributos, hiperplano, None, is_sigma=False)
        conjunto.append(atributos)
        if clases != None:
            soluciones.append(clases[prediccion])
        else:
            soluciones.append(prediccion)
    if not separable:
        soluciones = altera_soluciones(soluciones, rango)
    return (hiperplano, conjunto, soluciones)
```


Generador de conjuntos

- Luego es suficiente con generar puntos aleatorios y decidir por medio de alguna función de las estudiadas en qué parte estarían. Nosotros hemos elegido la función de predicción del perceptron.

```
def generar_conjunto_aleatorio(rango, dim, tam, separable=True, clases=None):  
    hiperplano = Clasificador.genera_pesos(dim)  
    conjunto = []  
    soluciones = []  
    for i in range(0, tam):  
        atributos = []  
        for j in range(0, dim):  
            atributos.append(random.randint(-rango, rango))  
        prediccion = Clasificador.calcular_prediccion(atributos, hiperplano, None, is_sigma=False)  
        conjunto.append(atributos)  
        if clases != None:  
            soluciones.append(clases[prediccion])  
        else:  
            soluciones.append(prediccion)  
    if not separable:  
        soluciones = altera_soluciones(soluciones, rango)  
    return (hiperplano, conjunto, soluciones)
```


Generador de conjuntos

- Para conseguir conjuntos linealmente no separables basta con alterar la clasificación de alguno de los puntos obtenidos anteriormente.

```
def altera_soluciones(originales, rango, porcentaje=0.2):  
    soluciones = copy.copy(originales)  
    percent = math.floor(len(originales) * porcentaje)  
    used = []  
    for i in range(0, percent):  
        index = random.randint(0, len(originales) - 1)  
        while index in used:  
            index = random.randint(0, len(originales) - 1)  
        used.append(index)  
        valor = random.randint(-rango, rango)  
        while valor == originales[index]:  
            valor = random.randint(-rango, rango)  
        soluciones[index] = valor  
    return soluciones
```

Pruebas neutrales

- ▶ Para las pruebas neutrales tenemos dos conjuntos de datos
 - Votos.py
 - Digidata

```
# El valor de cada votación lo codificamos numéricamente
# de la siguiente manera:
```

```
# 1: voto sí
# -1: voto no
# 0: "Present" (similar a la abstención)
```

```
votos_clases=['republicano','democrata']
```

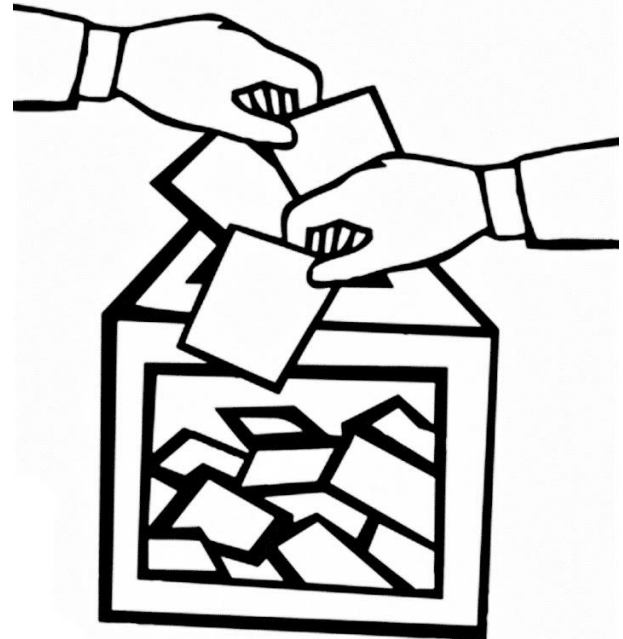
```
votos_entre= [[-1,1,-1,1,1,1,-1,-1,-1,1,0,1,1,1,-1,1],
               [-1,1,-1,1,1,1,-1,-1,-1,-1,-1,1,1,1,-1,0],
               [0,1,1,0,1,1,-1,-1,-1,-1,1,-1,1,1,-1,-1],
```

[illegible][illegible]

Pruebas neutrales

Votos

- ▶ Los atributos son varias votaciones del congreso de los estados unidos.
- ▶ Los valores pueden ser 1 para si, 0 para presente y -1 para no.
- ▶ Las posibles clases son republicanos o demócratas.



Pruebas neutrales

Votos

- ▶ Las pruebas con los votos nos dan estos resultados:

Método	Puntuación	Normalizado	Dacaimiento
Regresión estocástico	0.9710144927536232	NO	NO
Regresión batch	0.9710144927536232	NO	NO
Maximizar estocástico	0.9710144927536232	NO	SI
Regresión estocástico	0.9710144927536232	NO	SI
Regresión batch	0.9710144927536232	NO	SI
Maximizar estocástico	0.9710144927536232	SI	NO
Maximizar batch	0.9710144927536232	SI	NO
Regresión estocástico	0.9710144927536232	SI	NO
Regresión batch	0.9710144927536232	SI	NO

Pruebas neutrales

Votos

- ▶ Las pruebas con los votos nos dan estos resultados:

Método	Puntuación	Normalizado	Dacaimiento
Maximizar estocástico	0.9565217391304348	NO	NO
Maximizar batch	0.9565217391304348	NO	NO
Preceptrón estocástico	0.9420289855072463	NO	NO
Preceptrón estocástico	0.9420289855072463	NO	SI
Preceptrón estocástico	0.927536231884058	SI	NO
Preceptrón estocástico	0.927536231884058	SI	SI
Maximizar batch	0.8840579710144928	SI	SI
Maximizar batch	0.8695652173913043	NO	SI

Pruebas neutrales

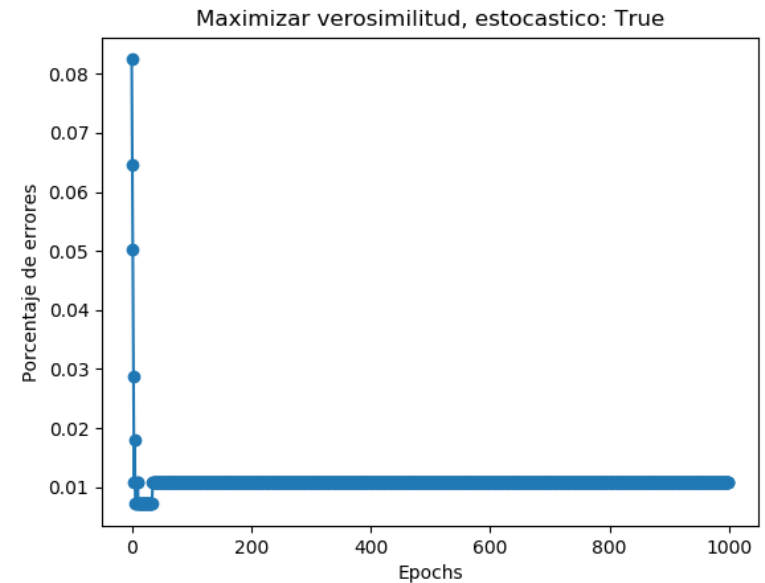
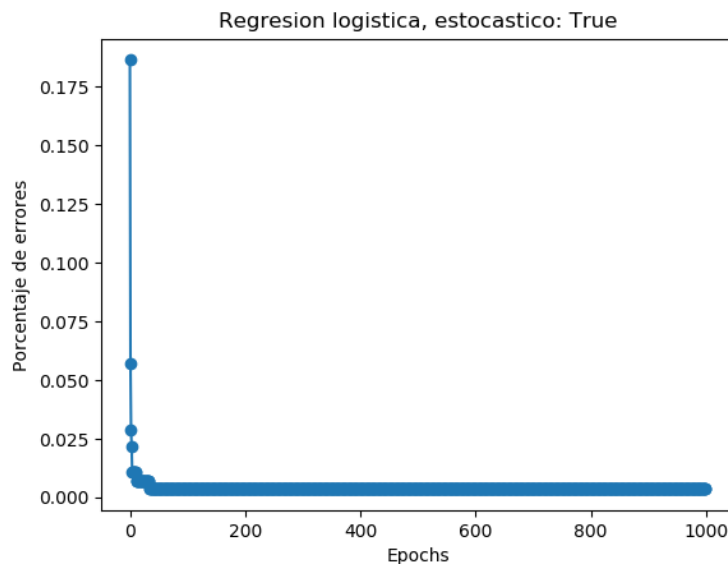
Votos

- ▶ El resultado es que estos métodos producen siempre los mejores resultados y convergen en el mismo porcentaje de acierto en el conjunto de los votos
- ▶ Minimizar error cuadrático
 - Estocástico y batch
 - Con o sin normalización
 - Con o sin decaimiento
- ▶ Maximizar log-verosimilitud
 - Estocástico
 - Con decaimiento y sin o con normalización
 - Sin decaimiento y con normalización
 - Batch
 - Sin decaimiento y con normalización

Pruebas neutrales

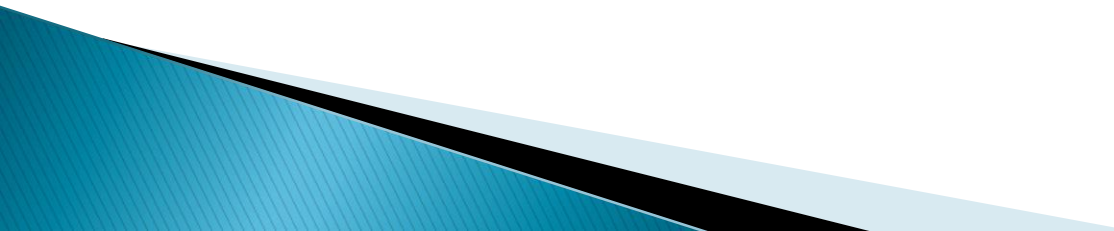
Votos

- ▶ Cualquiera de las opciones anteriores da los mismos resultados al final del entrenamiento.



Pruebas neutrales

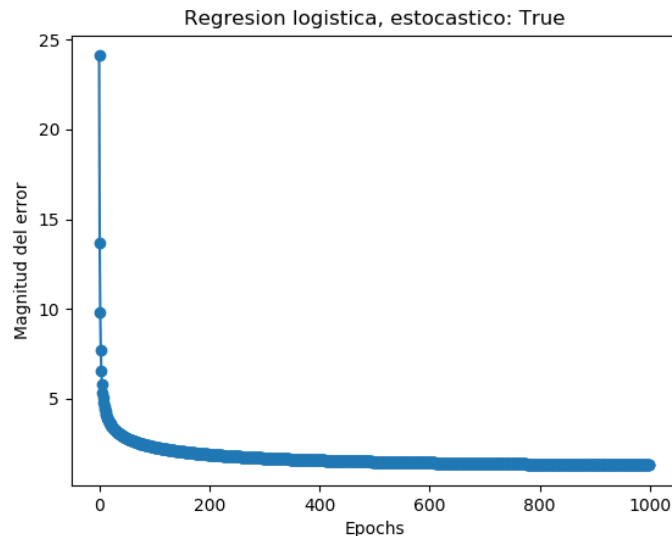
Votos

- ▶ Aunque convergen, hay elecciones mejores que otras.
 - ▶ La mejor elección es siempre depreciar el ratio de aprendizaje independientemente del resto de variables.
 - ▶ Normalizar no añade mejoras aparentes para este caso.
 - ▶ Las versiones estocásticas parecen comportarse mejor que las batchs aunque al final acaban convergiendo.
- 

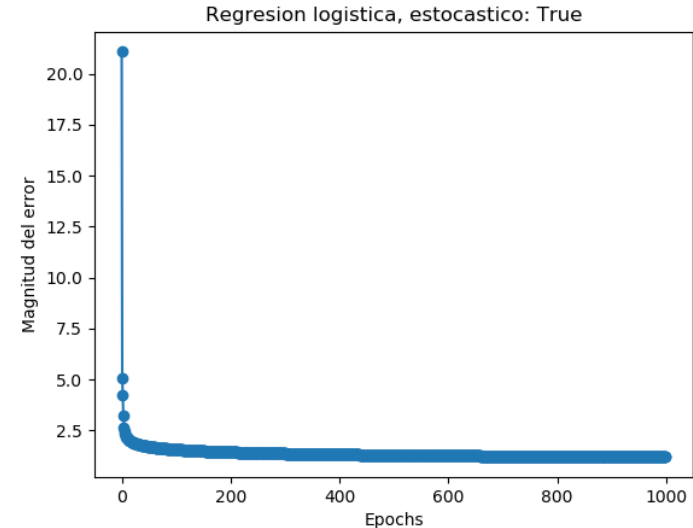
Pruebas neutrales

Votos

- Podemos comparar la minimización del error cuadrático.



- Estocástico
- Normalizado
- Sin deprecar

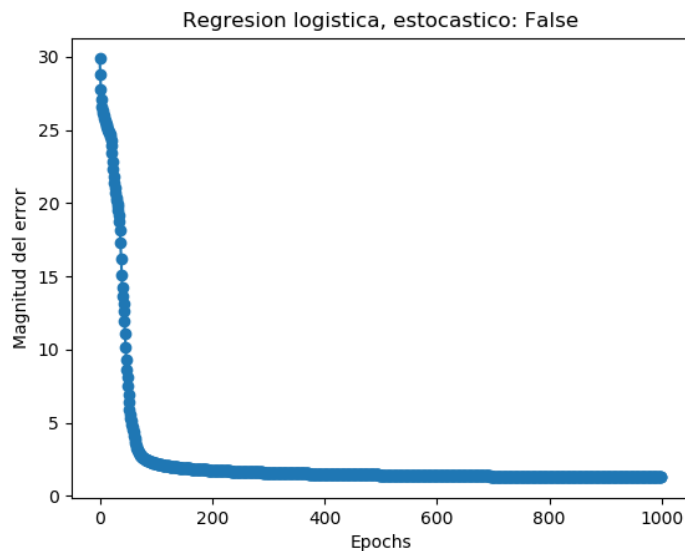


- Estocástico
- Normalizado
- Deprecando

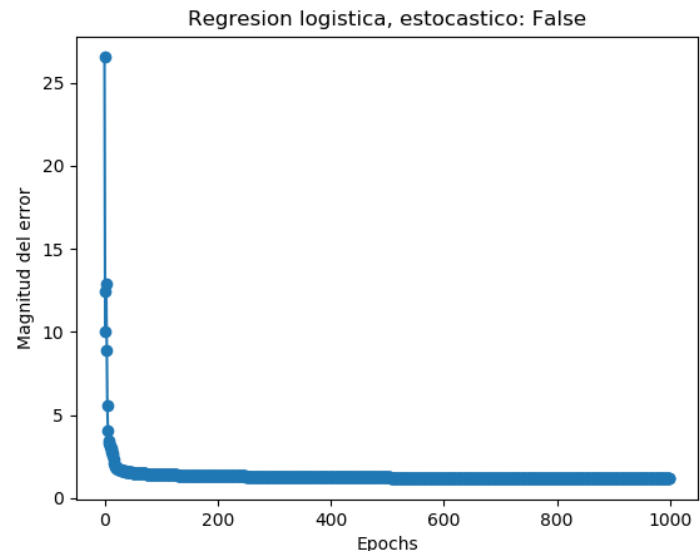
Pruebas neutrales

Votos

- Podemos comparar la minimización del error cuadrático.



- Batch
- Normalizado
- Sin deprecar

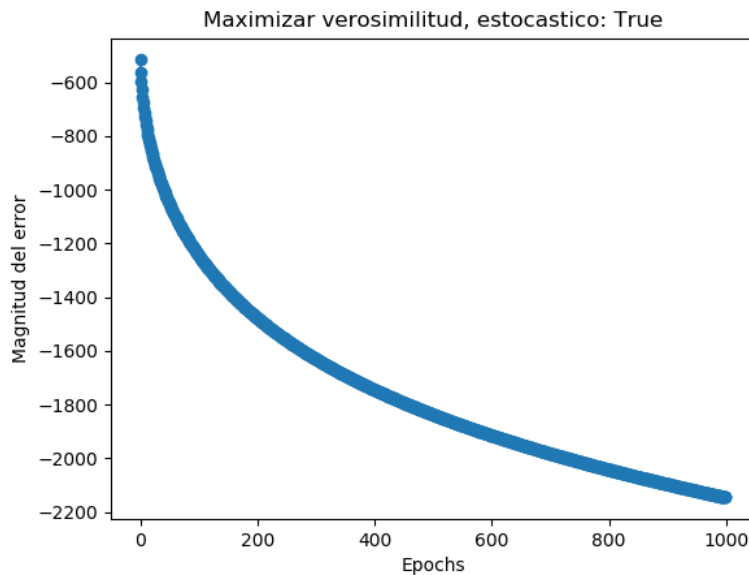


- Batch
- Normalizado
- Deprecando

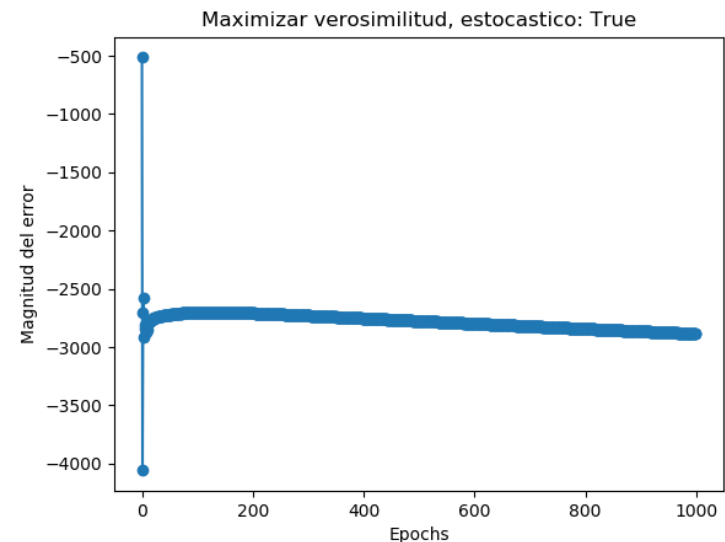
Pruebas neutrales

Votos

- ▶ Lo mismo ocurre para la maximización de la log-verosimilitud.



- Estocástico
- Normalizado
- Sin deprecar

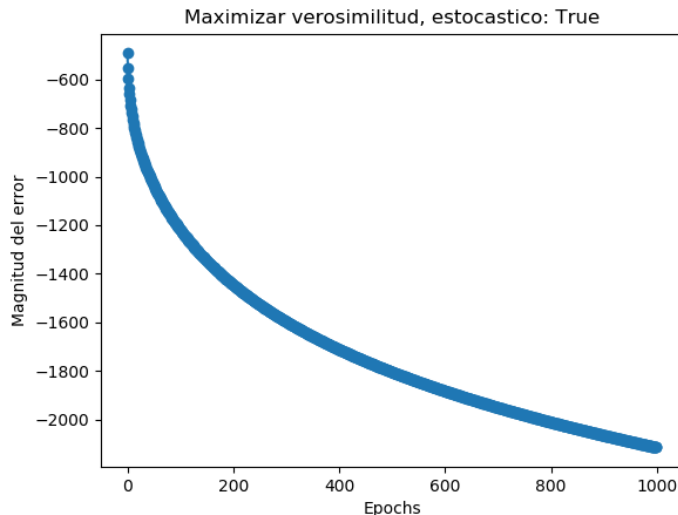


- Estocástico
- Normalizado
- Deprecando

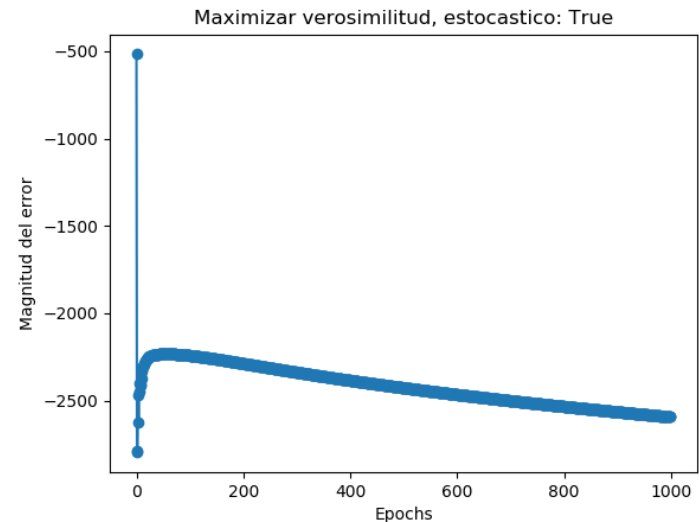
Pruebas neutrales

Votos

- ▶ Lo mismo ocurre para la maximización de la log-verosimilitud.



- Estocástico
- Sin normalizar
- Sin deprecar

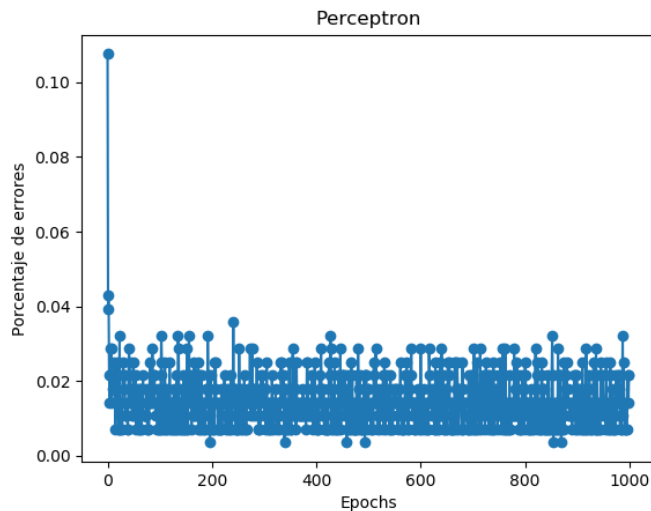


- Estocástico
- Sin normalizar
- Deprecando

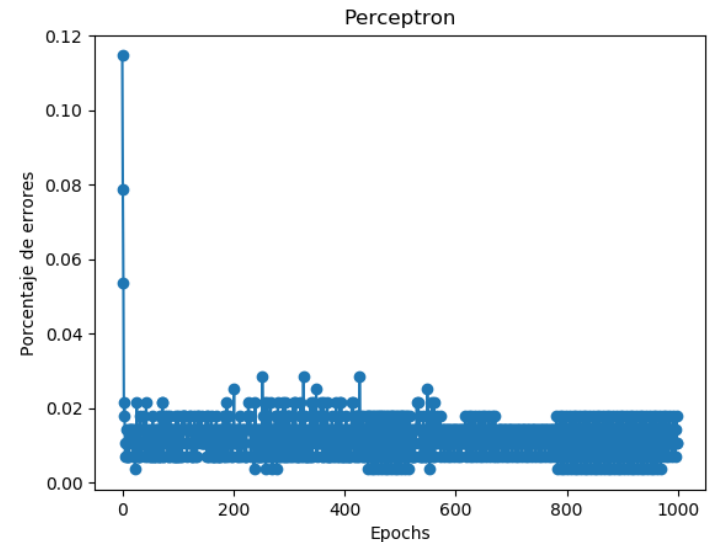
Pruebas neutrales

Votos

- ▶ Con el perceptrón vemos que el aprendizaje es errático pero acaba convergiendo.



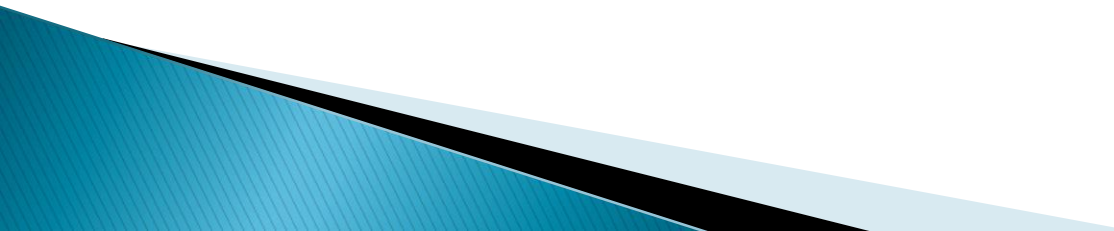
- Estocástico
- Normalizado
- Sin deprecar



- Estocástico
- Normalizado
- Deprecando

Pruebas neutrales

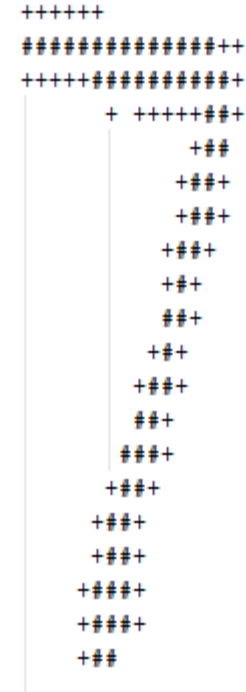
Votos

- ▶ Una excepción a todo lo dicho es la maximización batch con decaimiento.
 - ▶ Por alguna razón empeora mucho y no converge dando pesos muy elevados y malos.
 - ▶ Es probable que se deba a algún error de la implementación que no hemos sido capaces de detectar.
- 

Pruebas neutrales

Digitdata

- ▶ Son matrices de 27×27 que representan números del 0 al 9 escritos a mano.
- ▶ Cada atributo es una posición de la matriz,
- ▶ Los valores pueden ser 0 o 1 dependiendo se si hay algo escrito en la posición o no.
- ▶ Las posibles clases son los números del 0 al 9



Pruebas neutrales

Digitdata

- ▶ Como el grupo de datos total es muy grande y los ordenadores donde se han probado no terminaban nunca, se ha reducido la muestra para poder trabajar con ella.
- ▶ Conjunto de entrenamiento:
 - 2000 primeros ejemplos de números
- ▶ Conjunto de validación:
 - 500 primeros ejemplos de números
- ▶ Conjunto de test:
 - 500 primeros ejemplos de números

Pruebas aleatorias

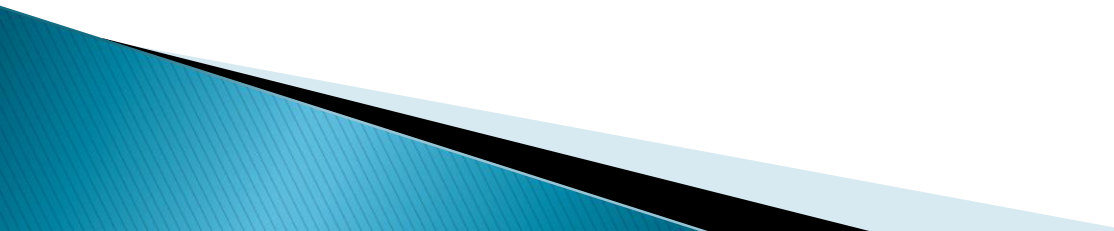
Votos

- ▶ Las pruebas aleatorias con los votos nos dan estos resultados:

Metodo	Validación	Prueba	Normalizado	Decaimiento
Regresion estocastico	0.98	0.985	NO	NO
Regresion batch	0.98	0.985	NO	NO
Maximizar estocastico	0.98	0.99	NO	SI
Regresion estocastico	0.98	0.985	NO	SI
Regresion batch	0.98	0.985	NO	SI
Maximizar estocastico	0.98	0.985	NO	NO
Maximizar batch	0.98	0.985	NO	NO
Preceptron estocastico	0.985	0.995	NO	NO
Preceptron estocastico	0.985	0.995	NO	SI
Maximizar batch	0.5	0.38	NO	SI

Pruebas aleatorias

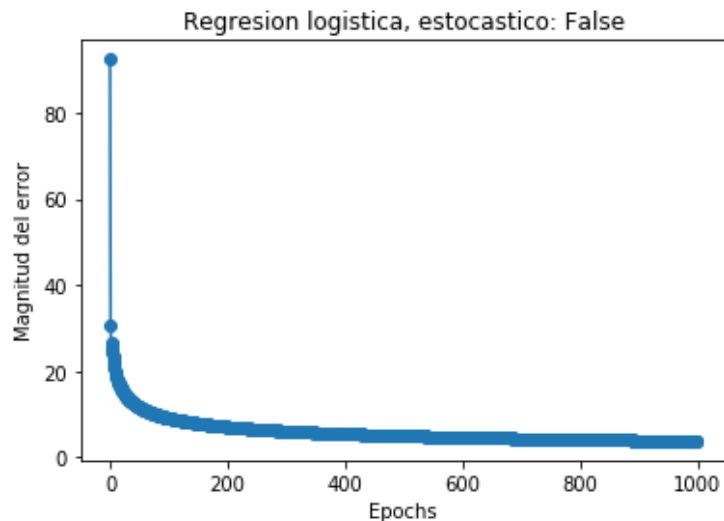
Votos

- ▶ Prácticamente para todos los casos tenemos un rendimientos bastante buenos y de similar valor.
 - ▶ Sin embargo, en la maximización por verosimilitud sin decaimiento tenemos un rendimiento muy pobre.
- 

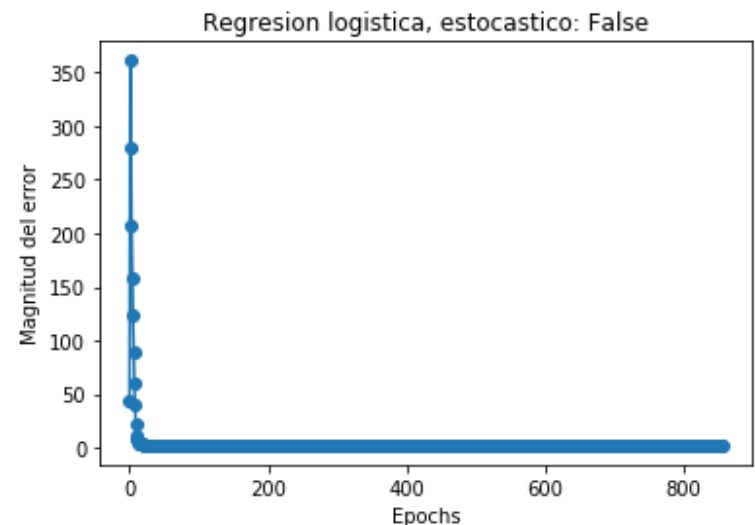
Pruebas aleatorias

Votos

- Podemos comparar la minimización del error cuadrático.



- Batch
- Sin normalizar
- Sin deprecar

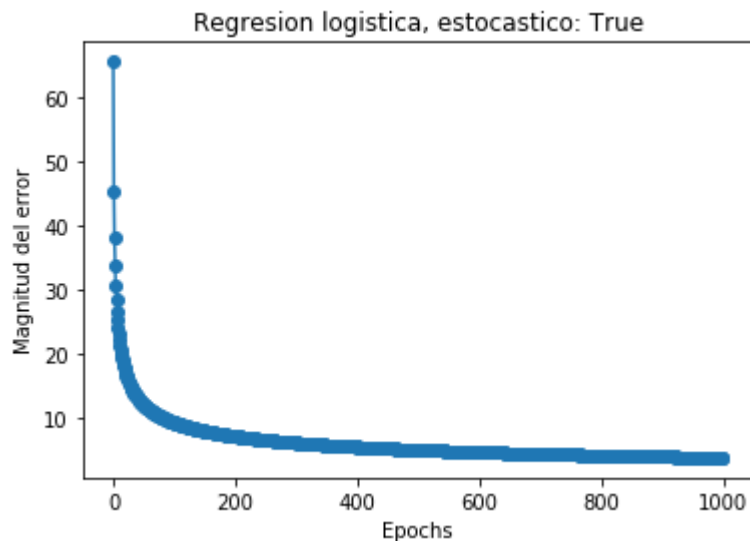


- Batch
- Sin normalizar
- Deprecando

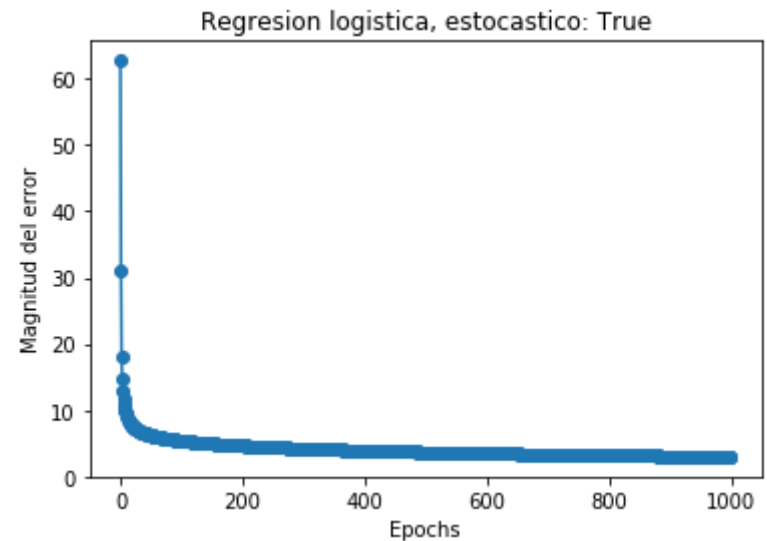
Pruebas aleatorias

Votos

- Podemos comparar la minimización del error cuadrático.



- Estocástico
- Sin normalizar
- Sin deprecar

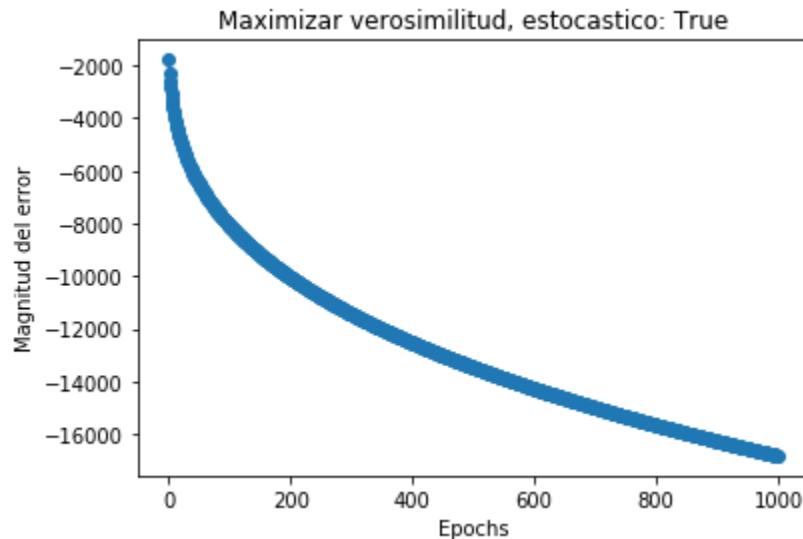


- Estocástico
- Sin normalizar
- Deprecando

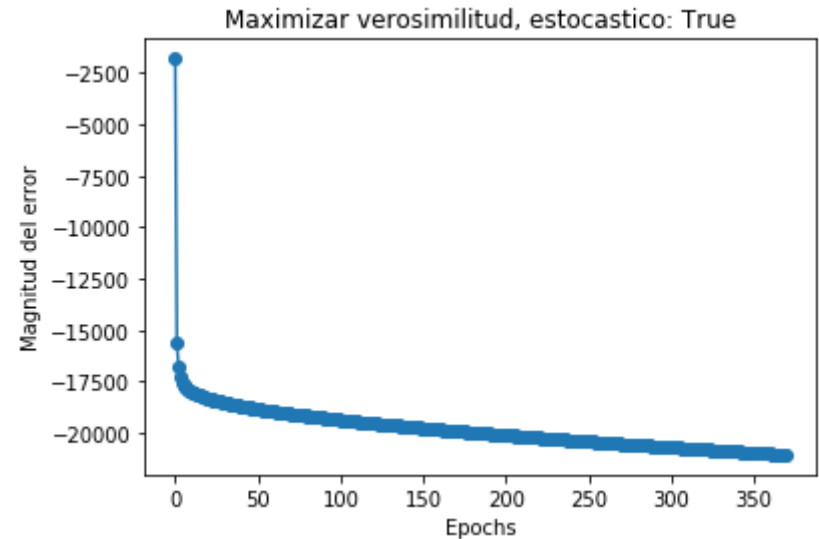
Pruebas aleatorias

Votos

- ▶ Lo mismo ocurre para la maximización de la verosimilitud.



- Estocástico
- Sin normalizar
- Sin deprecar



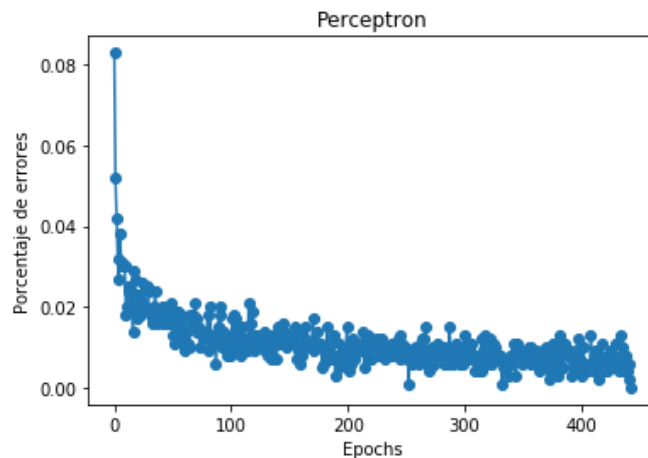
- Estocástico
- Sin normalizar
- Deprecando

Pruebas aleatorias

Votos

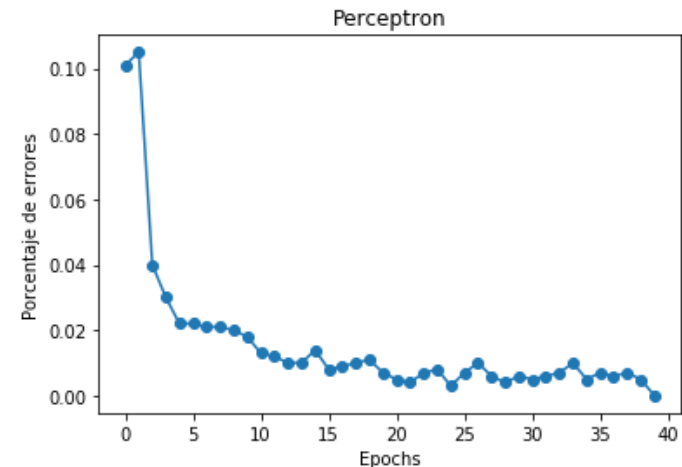
- ▶ Con el perceptrón vemos que el aprendizaje es errático pero también acaba convergiendo.

Sin decaimiento del ratio



- Sin normalizar
- Sin deprecar

Pruebas con decaimiento del ratio

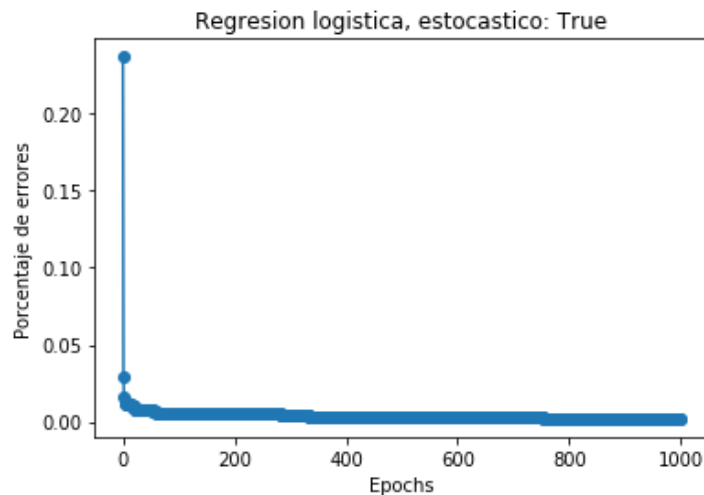


- Sin normalizar
- Deprecando

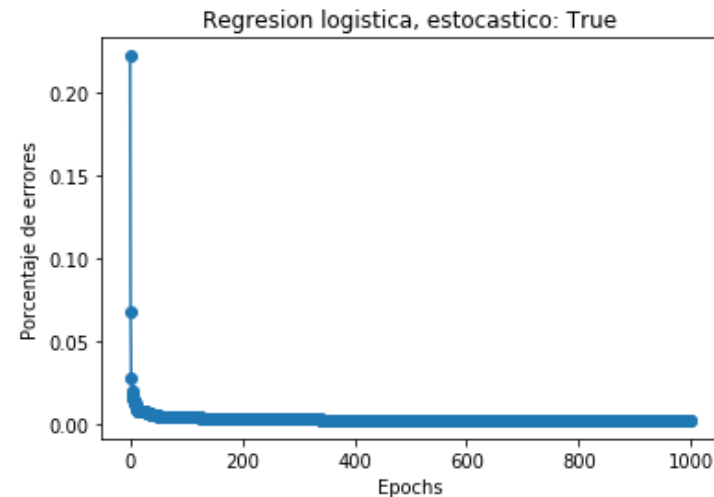
Pruebas aleatorias

Votos

- ▶ Con regresión vemos que el aprendizaje tiene curvas similares independientemente de si se aplica decaimiento o no.



- Estocástico
- Sin deprecar

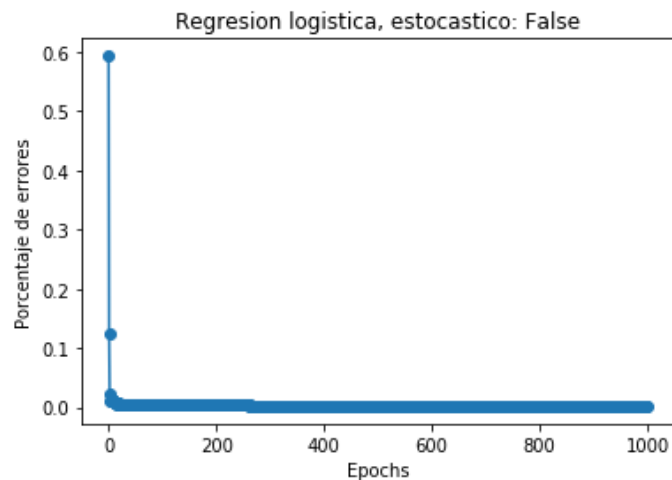


- Estocástico
- Deprecando

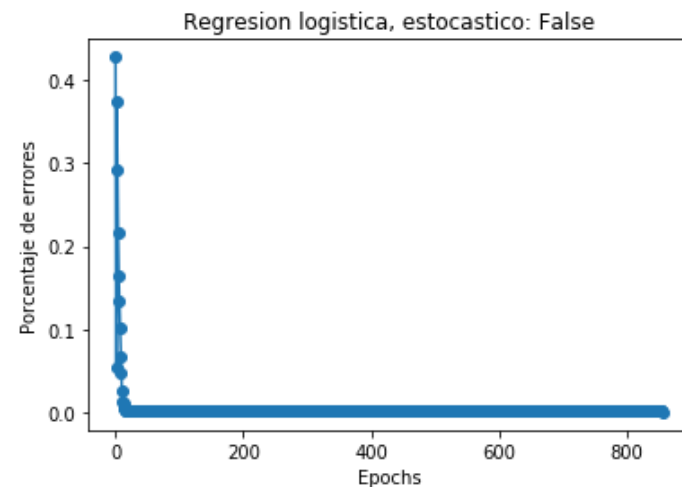
Pruebas aleatorias

Votos

- ▶ Con regresión vemos que el aprendizaje tiene curvas similares independientemente de si se aplica decaimiento o no.



- Batch
- Sin deprecar

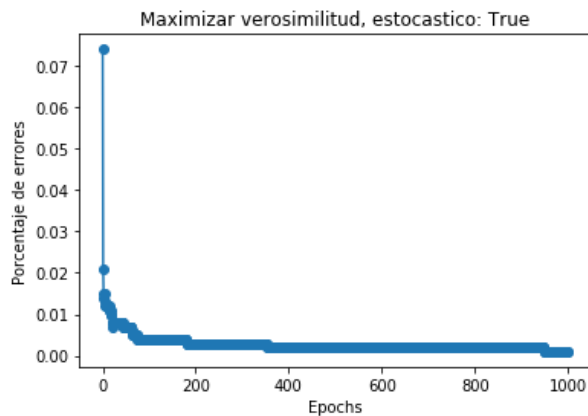


- Batch
- Deprecando

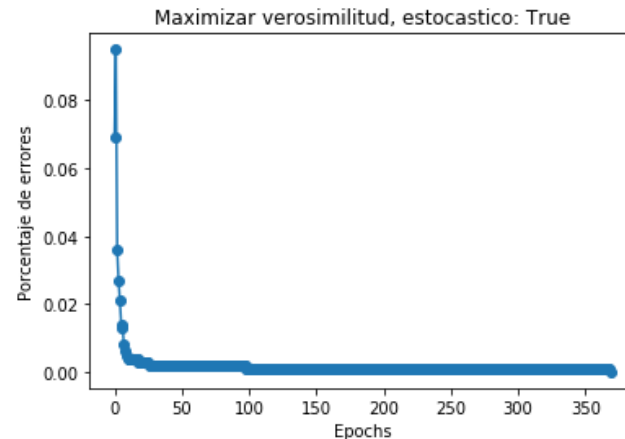
Pruebas aleatorias

Votos

- ▶ Para maximización de la verosimilitud se aprecian también curvas similares independientemente de la aplicación del decaimiento.



- Estocástico
- Sin deprecar



- Estocástico
- Deprecando

Pruebas aleatorias

Digitdata

- ▶ Las pruebas aleatorias con los dígitos nos dan estos resultados:

Metodo	Validación	Prueba	Normalizado	Decaimiento
Regresion estocastico	0.538	0.506	NO	NO
Regresion batch	0.085	0.075	NO	NO
Maximizar estocastico	0.625	0.55	NO	SI
Regresion estocastico	0.654	0.64	NO	SI
Regresion batch	0.085	0.075	NO	SI
Regresion estocastico	0.085	0.075	SI	NO
Regresion batch	0.085	0.075	SI	NO
Regresion estocastico	0.085	0.075	SI	SI
Regresion batch	0.085	0.075	SI	SI
Maximizar estocastico	0.635	0.6	NO	NO

Pruebas aleatorias

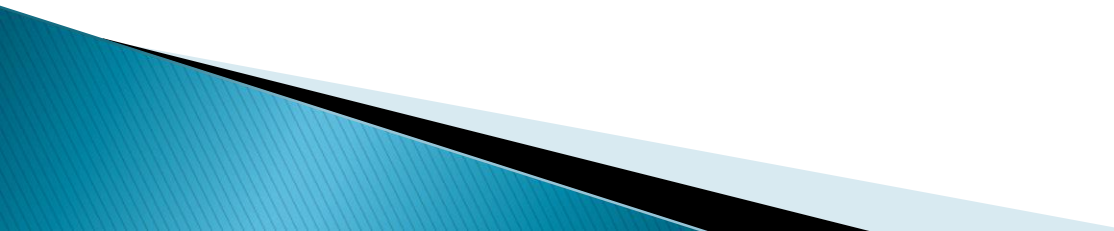
Digitdata

- ▶ Las pruebas aleatorias con los dígitos nos dan estos resultados:

Metodo	Validación	Prueba	Normalizado	Decaimiento
Maximizar batch	0.105	0.095	NO	NO
Preceptron estocastico	0.625	0.52	NO	NO
Preceptron estocastico	0.715	0.655	NO	SI
Preceptron estocastico	0.085	0.075	SI	NO
Preceptron estocastico	0.085	0.075	SI	SI
Maximizar batch	0.09	0.08	NO	SI

Pruebas aleatorias

Digitdata

- ▶ Prácticamente en la mayoría de casos el rendimiento es bastante bajo debido a que se usa un conjunto de entrenamiento reducido con respecto al original, ya que el tiempo de ejecución es muy grande.
 - ▶ No se han realizado pruebas para la maximización de la verosimilitud con normalización debido a errores en la ejecución, posiblemente debido al tratamiento de numpy con algunos números.
- 

2ª Parte – cáncer de mama

- ▶ Para esta parte se han implementado los siguientes clasificadores:
 - Knn
 - Perceptrón (SGDC)
 - Regresión logística (SGDC y LogisticRegression)
 - Vectores soporte (SGDC y LinearSVC)
 - Árboles de decisión
 - Random Forest

Se ha probado el rendimiento de ellos de distintas formas a partir del conjunto de datos de cáncer de mama.

2ª Parte – Knn

Cáncer de mama

- ▶ Se han probado distintas combinaciones de parámetros para este clasificador:
 - Dependiendo de si los datos se normalizan o no, en los que normalmente se suele observar una mejora en el rendimiento.
 - Dependiendo del número de vecinos (se ha usado desde $n=3$ hasta $n=9$).
 - Dependiendo de si el peso que se le da a los vecinos es uniforme o en base a la distancia (se hace dentro de la función Knn).

Realizando un total de 28 pruebas.

```
parametroNorm = [True, False]  
parametroNeighbors = [3,4,5,6,7,8,9]
```

2ª Parte – Knn

Cáncer de mama

- El mejor rendimiento se ha obtenido con normalización y $n=9$, con un rendimiento de 0.99300699300699302 para ambos pesos.

```
[[True,
  9,
  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                        metric_params=None, n_jobs=1, n_neighbors=9, p=2,
                        weights='distance'),
  0.99300699300699302],
 [True,
  9,
  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                        metric_params=None, n_jobs=1, n_neighbors=9, p=2,
                        weights='uniform'),
  0.99300699300699302]]
```

2ª Parte – Perceptrón

Cáncer de mama

- ▶ Se han probado distintas combinaciones de parámetros para este clasificador:
 - Dependiendo del número de epochs.
 - Dependiendo de la regularización que se aplica: L1, L2 o ninguna.
 - Dependiendo del valor del parámetro alpha que multiplica la regularización.
 - Dependiendo de si se aplica o no la tasa de aprendizaje.
 - Dependiendo del valor de la tasa de aprendizaje en caso de aplicarse.

Se han realizado 720 pruebas en este caso.

```
parametroEpoch = [10,100,500,1000,5000,10000]  
parametroPenalty = ["none","l1","l2"]  
parametroAlpha = [0.001,0.01,0.1,0.5,0.7]  
parametroTasaAp = [0.001,0.01,0.1,0.5]  
parametroAplicar = [True, False]
```


2ª Parte – Perceptrón

Cáncer de mama

- ▶ El mejor rendimiento se ha obtenido con epoch=10000, regularización ridge (L2), $\alpha = 0.001$ y sin aplicar tasa de aprendizaje, con un rendimiento del 0.97202797202797198.

```
[10000, 'l2', 0.001, 0.01, False, 0.97202797202797198]
```

2ª Parte – Regresión logística

Cáncer de mama

- ▶ En este caso, compararemos el rendimiento del clasificador en SGDC y LogisticRegression.
- ▶ En el caso de SGDC se han usado los mismos valores para los parámetros que en el caso anterior.
- ▶ Para LogisticRegression se han considerado los parámetros epoch, regularización y valor de C.

Para LogisticRegression se han realizado 60 pruebas, más las 720 de SGDC.

```
parametroEpoch = [10,100,500,1000,5000,10000]  
parametroPenalty = ["l1","l2"]  
parametroC = [0.0001,0.001,0.005,0.01,0.1]
```

2ª Parte – Regresión logística

Cáncer de mama

- ▶ Para SGDC se ha obtenido el siguiente rendimiento con epoch = 10000, sin regularización, con alpha = 0.5 y sin aplicar tasa.

```
[10000, 'none', 0.5, 0.01, True, 0.98601398601398604]
```

2ª Parte – Regresión logística

Cáncer de mama

- ▶ Para LogisticRegression se ha obtenido el siguiente rendimiento con epoch = 1000, con regularización L2, con $C = 0.1$.

```
[1000, 'l2', 0.1, 0.965034965034965]
```

- ▶ Se aprecia un mejor rendimiento con SGDC, sin embargo en LogisticRegression se han necesitado un menor número de epoch y pruebas con un rendimiento bastante bueno.

2ª Parte – Vectores soporte

Cáncer de mama

- ▶ En este caso, compararemos el rendimiento del clasificador en SGDC y LinearSVC.
- ▶ En el caso de SGDC se han usado los mismos valores para los parámetros que los casos anteriores.
- ▶ Para LinearSVC se han considerado los parámetros epoch, loss (si se usa “hinge” o “squared hinge”) y valor de C.

Para LinearSVC se han realizado 60 pruebas, más las 720 de SGDC.

```
parametroEpoch = [10,100,500,1000,5000,10000]  
parametroLoss = [True, False]  
parametroC = [0.0001,0.001,0.005,0.01,0.1]
```

2ª Parte – Vectores soporte Cáncer de mama

- ▶ Para SGDC se ha obtenido el siguiente rendimiento con epoch = 500, sin regularización, con alpha = 0.7 y aplicando una tasa de aprendizaje de 0.1.

```
[500, 'none', 0.7, 0.1, True, 0.97902097902097907]
```

2ª Parte – Vectores soporte Cáncer de mama

- ▶ Para LinearSVC se ha obtenido el siguiente rendimiento con epoch = 100, aplicando squared hinge con C= 0.0001.

```
[100, 0.0001, False, 0.97202797202797198]
```

- ▶ Se aprecia un mayor rendimiento en SGDC pero no por mucho, considerando el número de pruebas en LinearSVC y el menor número de epoch.

2ª Parte – Árboles de decisión

Cáncer de mama

- ▶ En este caso los parámetros que consideraremos son la profundidad del árbol, el criterio que se aplicará y el mínimo de ejemplos.

```
parametroProfundidad = [2,3,4,5,6,7,8,9,10]  
parametroCriterio = ["gini", "entropy"]  
parametroMinEjemplos = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
```

- ▶ Se realizan 162 pruebas en este caso.

2ª Parte – Árboles de decisión

Cáncer de mama

- ▶ El mejor rendimiento que se ha encontrado es de 0.965034965034965 para un valor de profundidad = 7, aplicando entropía y con un mínimo de ejemplos de 0.1.

```
[7, 'entropy', 0.1, 0.965034965034965]
```

2ª Parte – Random forest

Cáncer de mama

- ▶ Para este caso se han considerado los mismos valores y parámetros, añadiendo uno nuevo para aplicar o no bootstrap para evitar el sobreajuste.

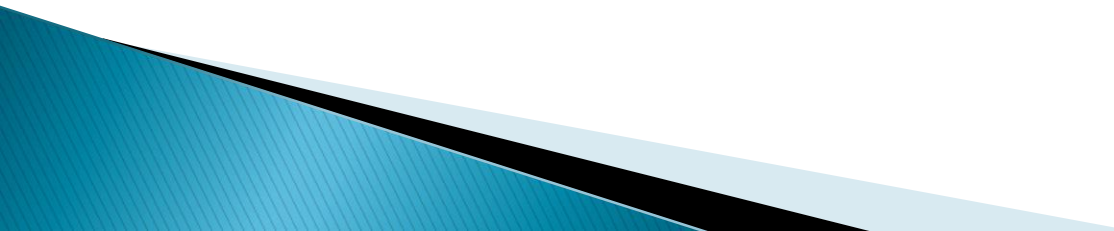
```
parametroProfundidad = [2,3,4,5,6,7,8,9,10]  
parametroBootstrap = [True, False]  
parametroCriterio = ["gini", "entropy"]  
parametroMinEjemplos = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
```

2ª Parte – Random forest

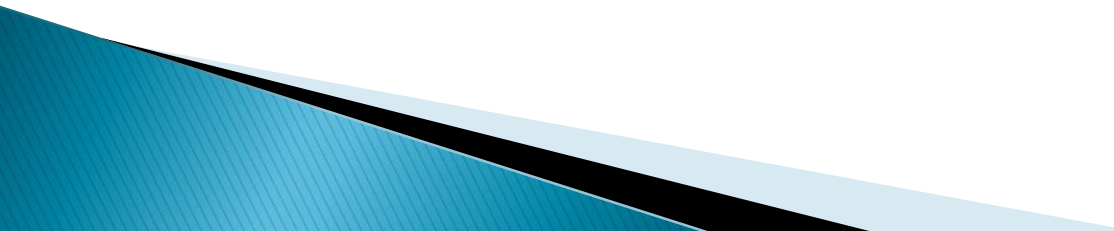
Cáncer de mama

- ▶ Se ha obtenido un mejor rendimiento que en el caso de los árboles de decisión, con una menor profundidad y aplicando los mismos parámetros.

```
[4, False, 'entropy', 0.1, 0.97902097902097907]
```



2ª Parte – Digitdata

- ▶ Para este caso se han ejecutado los mismos clasificadores con los mismos posibles valores para los parámetros de entrada.
 - ▶ Se ha usado la librería de numpy para pasar los datos de dígitos a arrays de numpy y poder manipularlos en scikit.
 - ▶ Se han usado 100 ejemplos para entrenamiento y 25 para prueba.
- 

2ª Parte – Knn

Digitdata

- ▶ Para este caso no se han normalizado los datos debido a que todas las columnas tienen el mismo rango de valores. Los demás parámetros se han mantenido:
 - Dependiendo del número de vecinos (se ha usado desde $n=3$ hasta $n=9$).
 - Dependiendo de si el peso que se le da a los vecinos es uniforme o en base a la distancia (se hace dentro de la función Knn).

Se realiza un total de 14 pruebas.

2ª Parte – Knn

Digitdata

- ▶ Se han obtenido rendimientos altos para un número de vecinos = 3, aunque se aprecia un mayor rendimiento cuando el peso a los vecinos depende de la distancia.

```
[[3, KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',  
    metric_params=None, n_jobs=1, n_neighbors=3, p=2,  
    weights='distance'), 1.0], [3, KNeighborsClassifier(algorithm='auto',  
leaf_size=30, metric='minkowski',  
    metric_params=None, n_jobs=1, n_neighbors=3, p=2,  
    weights='uniform'), 0.95999999999999996]]
```

2ª Parte – Perceptrón

Digitdata

- ▶ Se han probado distintas combinaciones de parámetros para este clasificador:
 - Dependiendo del número de epochs.
 - Dependiendo de la regularización que se aplica: L1, L2 o ninguna.
 - Dependiendo del valor del parámetro alpha que multiplica la regularización.
 - Dependiendo de si se aplica o no la tasa de aprendizaje.
 - Dependiendo del valor de la tasa de aprendizaje en caso de aplicarse.

Se han realizado 720 pruebas en este caso.

2ª Parte – Perceptrón

Digitdata

- ▶ Se han observado varias ejecuciones en las que el rendimiento es del 100%. Una de las combinaciones de valores con las que se obtiene dicho rendimiento es $\text{epoch} = 10$, sin aplicar regularización y aplicando tasa de aprendizaje = 0.001.

```
[10, 'none', 0.001, 0.001, True, 1.0]
```


2ª Parte – Regresión logística

Digitdata

- ▶ En este caso, compararemos el rendimiento del clasificador en SGDC y LogisticRegression.
- ▶ En el caso de SGDC se han usado los mismos valores para los parámetros que los casos anteriores, excepto para los valores 5000 y 10000 de epoch para reducir el tiempo de ejecución.
- ▶ Para LogisticRegression se han considerado los parámetros epoch, regularización y valor de C.

Para LogisticRegression se han realizado 60 pruebas, más las 720 de SGDC.

```
parametroEpoch = [10,100,500,1000,5000,10000]  
parametroPenalty = ["l1","l2"]  
parametroC = [0.0001,0.001,0.005,0.01,0.1]
```

2ª Parte – Regresión logística

Digitdata

- ▶ Para SGDC se ha obtenido el siguiente rendimiento con epoch = 10, sin aplicar regularización ni tasa de aprendizaje.

```
[10, 'none', 0.001, 0.001, False, 1.0]
```

2ª Parte – Regresión logística

Digitdata

- ▶ Para LogisticRegression se ha obtenido el siguiente rendimiento con $\text{epoch} = 10$, con regularización L2, con $C = 0.1$.

`[1.0, 'l2', 0.1, 1.0]`

- ▶ En ambos casos el rendimiento es del 100% para el conjunto de prueba con el mismo número de epochs.

2ª Parte – Vectores soporte

Digitdata

- ▶ En este caso, compararemos el rendimiento del clasificador en SGDC y LinearSVC.
- ▶ En el caso de SGDC se han usado los mismos valores para los parámetros que los casos anteriores, excepto para los valores 5000 y 10000 de epoch para reducir el tiempo de ejecución.
- ▶ Para LinearSVC se han considerado los parámetros epoch, loss (si se usa “hinge” o “squared hinge”) y valor de C.

Para LinearSVC se han realizado 60 pruebas, más las 720 de SGDC.

```
parametroEpoch = [10,100,500,1000,5000,10000]  
parametroLoss = [True, False]  
parametroC = [0.0001,0.001,0.005,0.01,0.1]
```

2ª Parte – Vectores soporte Digitdata

- ▶ Para SGDC se ha obtenido el siguiente rendimiento con epoch = 10, sin aplicar regularización ni tasa de aprendizaje.

```
[10, 'none', 0.001, 0.001, False, 1.0]
```

2ª Parte – Vectores soporte Digitdata

- ▶ Para LinearSVC se ha obtenido el siguiente rendimiento con epoch = 10, aplicando hinge con $C = 0.1$.

```
[10, 0.1, True, 1.0]
```

- ▶ En ambos casos el rendimiento es del 100% para el mismo número de epochs.

2ª Parte – Árboles de decisión

Digitdata

- ▶ En este caso los parámetros que consideraremos son la profundidad del árbol, el criterio que se aplicará y el mínimo de ejemplos.

```
parametroProfundidad = [2,3,4,5,6,7,8,9,10]  
parametroCriterio = ["gini", "entropy"]  
parametroMinEjemplos = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
```

- ▶ Se realizan 162 pruebas en este caso.

2ª Parte – Árboles de decisión

Digitdata

- ▶ El mejor rendimiento que se ha encontrado es de 0.920000000000000000004 para un valor de profundidad = 5, aplicando entropía y con un mínimo de ejemplos de 0.1.

```
[5, 'entropy', 0.1, 0.920000000000000000004]
```


2ª Parte – Random forest

Digitdata

- ▶ Para este caso se han considerado los mismos valores y parámetros, añadiendo uno nuevo para aplicar o no bootstrap para evitar el sobreajuste.

```
parametroProfundidad = [2,3,4,5,6,7,8,9,10]  
parametroBootstrap = [True, False]  
parametroCriterio = ["gini", "entropy"]  
parametroMinEjemplos = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
```

2ª Parte – Random forest

Digitdata

- ▶ Se ha obtenido un mejor rendimiento que en el caso de los árboles de decisión, con una menor profundidad, aplicando Gini y sin aplicar bootstrapping.

```
[6, False, 'gini', 0.1, 1.0]
```