# Modelización predictiva modelo supervisado por k vecinos

Contenido

[**A) DateSet Digits 2**](#_Toc505939460)

[**Ejercicio 1: Descripción de campos 2**](#_Toc505939461)

[**Ejercicio 2: Descripción del dataset 5**](#_Toc505939462)

[**Ejercicio 3: Experimentos vistos en clase 5**](#_Toc505939463)

**Ejercicio 4: Comparación Leave-One-Out y CrossValidation** [**14**](#_Toc505939464)

[**Ejercicio 5: Estratificación del proceso de validación 22**](#_Toc505939465)

[**Ejercicio 6: Ajuste de los pesos de la métrica de distancia 23**](#_Toc505939466)

[**Ejercicio 7: Ajuste del tipo de métrica de distancia 25**](#_Toc505939467)

### DESCRIPCION DEL DATASET

El dataset contiene imágenes de números escritos a mano.

Este dataset describe pixeles de imágenes codificados con valores numéricos del 0 al 16. Consta de 5620 registros y tiene 64 características para cada registro. Las labels son valores numéricos del 0 al 9.

Notes

-----

Los datos se han extraído y normalizado a partir de mapas de bit de dígitos escritos a mano. Los mapas de bits de 32x32 se dividen

en bloques no superpuestos de 4x4 y se cuenta el número de pixeles en cada bloque, esto genera una matriz de 8x8 de números enteros en un rango del 1 al 16

Data Set Characteristics:

   :Number of Instances: 5620

   :Number of Attributes: 64

   :Attribute Information: 8x8 image of integer pixels in the range 0..16.

   :Missing Attribute Values: None

   :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)

   :Date: July; 1998

### Experimentos con KNN

#### KNN1

Aplicación del modelo k-vecinos habiendo estratificado y dividido el dataset para aplicar una validación con la función StratifiedShuffleSplit().

from sklearn.datasets import load\_digits

from sklearn.neighbors import KNeighborsClassifier

import numpy as np

import matplotlib.pyplot as plt

datadigits = load\_digits()

X= datadigits.data

Y= datadigits.target

from sklearn.model\_selection import StratifiedShuffleSplit

**#importa el método de la librería**

**#Este método provee de un índice de train y test para partir el dataset en sus partes correspondientes**.

**# StratifiedShuffleSplit(self, n\_splits=10, test\_size="default", train\_size=None, random\_state=None)**

**#self: default=10, número de re-shufling y splitting iterations**

**#test\_size: Opcional. float ente 0.0 y 1.0. Representa la proporción del dataset incluida en el corte de test**

**#train\_size: Opcional. float entre 0.0 y 1.0. Representa la proposción del dataset incluida en el corte de train**

**#random\_state: default=None. Es la semilla utilizada por el generador de números random.**

n=1 **#o 20 para el otro ejemplo**

misss=StratifiedShuffleSplit(n,0.3) **#n cortes y una proporción del 30% de la parte de test**

fallos=[] **#inicializa un array para guardar la suma de los fallos de la predicción**

index=0 **#inicializa una variable para el índice**

for train\_index, test\_index in misss.split(X, Y): **#inicia un bucle for: para el índice de train y el de test en el método de corte StratifiedShuffleSplit**

print (index) **#imprime por pantalla el indice**

print (train\_index) **#imprime por pantalla el índice de train**

print (test\_index) **#imprime por pantalla el índice de index**

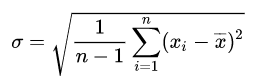
Xtrain=X[train\_index,:] **#crea una matriz con los datos de train. Utilizando el índice obtenido**

Xtest=X[test\_index,:] **#Hace lo mismo con la parte de test**

Ytrain=Y[train\_index] **#Y repite la operación con el vector Y de train**

Ytest=Y[test\_index] **#y de test**

miKvecinos=KNeighborsClassifier(n\_neighbors=3)



**#implementa el clasificador del voto de los k vecinos más cercanos**

**#KNeighborsClassifier(n\_neighbors=5, weights='uniform', algorithm='auto', leaf\_size=30, p=2, metric='minkowski', metric\_params=None, n\_jobs=1, \*\*kwargs)**

**#n\_neighbors: Opcional, default=5 número de vecinos**

**#algorithm: Opcional, el algoritmo utilizado para computar los vecinos más cercanos:**

**#ball\_tree -> BallTree**

**#kd\_tree -> KDTree**

**#brute ->usara fuerza bruta para la búsqueda**

**#auto -> utilizará el que considere más apropiado**

miKvecinos.fit(Xtrain,Ytrain) **#entrenas el modelo con la matriz X y las etiquetas de entrenamiento**

Ypred=miKvecinos.predict(Xtest) **#obtienes una predicción con la parte reservada para el test**

fallos.append(sum(Ypred!=Ytest)) **#un contador de los fallos en la predicción**

index=index+1 **#aumenta el índice, pasando al siguiente y avanzando el bucle**

print("fallos: "+str(fallos))

print ("Num. medio de errores de: " + str(100\*np.mean(fallos)/len(Ytest))) **#imprime por pantalla la media**

print ("Dev. Std. de errores de: " + str(100\*np.std(fallos)/len(Ytest))) **#imprime por pantalla la desviación**

Aplicamos la función **StratifiedShuffleSplit** con sólo **un corte(Split)** y seleccionando un 30% del dataset para el test :

Num. medio de errores de: 1.2962962963

Dev. Std. de errores de: 0.0

El número medio de errores al sólo haber hecho un corte no es un dato a tener en cuenta ya que dependiendo de la muestra de validación puede variar mucho el resultado.

Al haber hecho solo un corte aleatorio en el dataset el resultado de la media y la desviación no es un dato relevante ya que depende de la distribución original de los datos dentro del dataset.

La desviación es nula por la misma razón.

En este caso el numero medio de errores es un dato bastante optimo, es un dataset grande y su distribución es equilibrada.

Usando la función **StratifiedShuffleSplit** con **20** para el número de cortes y 0.3 para el test.

*Resultado*

*Num. medio de errores de: 1.28703703704*

*Dev. Std. de errores de: 0.550828656212*

**Conclusión:**

Los resultados sobre el número medio de errores no varían mucho haciendo un solo Split o haciendo 20, por lo que podemos deducir que la distribución del dataset es buena, los datos están muy mezclados.

**Grafica de los resultados**

**#Grafica con la media de los errores y la desviacion dependiendo**

**#del numero de splits que apliquemos y con kvecinos=3**

desv=[]

error=[]

for n in range(1,30):

misss=StratifiedShuffleSplit(n,0.3) #20 cortes y una proporción del 30% de la parte de test

fallos=[]

index=0

for train\_index, test\_index in misss.split(X, Y):

Xtrain=X[train\_index,:]

Xtest=X[test\_index,:]

Ytrain=Y[train\_index]

Ytest=Y[test\_index]

miKvecinos=KNeighborsClassifier(n\_neighbors=3)

miKvecinos.fit(Xtrain,Ytrain)

Ypred=miKvecinos.predict(Xtest)

fallos.append(sum(Ypred!=Ytest))

index=index+1

error.append(100\*np.mean(fallos)/len(Ytest))

desv.append(100\*np.std(fallos)/len(Ytest))

plt.plot(range(1,30),error, 'b-', label='Num. medio de errores')

plt.plot(range(1,30), desv, 'r-', label='Desv. Std de errores')

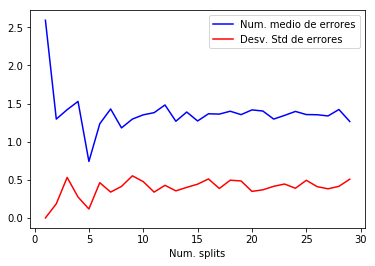
plt.legend(loc='upper right')

plt.xlabel('Num. splits')

plt.title('Análisis media/varianza/splits')

plt.ylim(0,4)

plt.style.context('seaborn-whitegrid')



Al visualizar los datos en la gráfica se observa que a partir de 5 splits los datos de la media y desviación empiezan a estabilizarse. Seria optimo usar un numero de splits mayor de 5 para obtener resultados con este dataset con el modelo de KNN.

#### KNN2

Aplicación del modelo cross\_val\_score que efectúa una crosvalidación al dataset utilizando K-vecinos como método y devuelve medidas de los resultados como la media o la desviación.

Usamos el método kvecinos con 3 vecinos como parámetro.

Este método particiona el dataset en: datos de training y datos de test, entrena el método sobre los datos de training y después valida el método sobre los datos de test.

El parámetro del numero de crossvalidaciones indica el número de particiones que vamos a hacer al dataset y el numero de veces que se va a repetir el proceso.

Si el parámetro es N, particionaremos el dataset en N subconjuntos, entrenaremos con N-1 subconjuntos y validaremos el dataset sobre el subconjunto restante repitiendo el proceso N veces.

from sklearn.datasets import load\_digits

from sklearn.neighbors import KNeighborsClassifier

import numpy as np

misdatos= load\_digits () **#cargo la base de datos en una variable**

X=misdatos.data **#creo la matriz X con los datos**

Y=misdatos.target **#creo la matriz Y con las etiquetas**

miKvecinos=KNeighborsClassifier(n\_neighbors=3)

from sklearn.model\_selection import cross\_val\_score **#importa el modelo cross\_val\_score**

**#cross\_val\_score(estimator, X, y=None, groups=None, scoring=None, cv=None, n\_jobs=1, verbose=0, fit\_params=None, pre\_dispatch='2\*n\_jobs')**

**#Evalúa una puntuación para una cross-validation**

**#estimator: el modelo al que implementar fit**

**#X: la matriz de los datos en un array**

**#Y: las etiquetas en caso de aprendizaje supervisado**

**#groups:**

**#scoring: una cadena o un puntuador ¿**

**#cv: deternina la crossvalidación: puede ser un entero (el valor por defecto es 3) un objeto o un iterable**

**#n\_jobs: el número de CPU que usa la computación**

**#verbose: nivel de verbosidad**

**#fit\_params : parámetros que se pan al método fit.**

**#pre\_dispatch: controla el número de trabajos durante una ejecución paralela.**

micvs=cross\_val\_score(miKvecinos,X,Y,cv=10) #crossvalida 10 veces usando kvecinos

print(micvs)

print("Mean:" + str(np.mean(micvs))) **#saca la media de todas las crossvalidaciones**

print("Std: "+ str(np.std(micvs))) **#saca la varianza de todas las crossvalidaciones**

Mean:0.98244994406

Std:0.0161038114221

**Conclusión**

Con el método de CV evitamos sobreajuste ya que el subconjunto de datos que forma el training varía en cada iteración.

Al aplicar el método k-vecinos y 10 particiones el número de errores disminuye considerablemente y la varianza es más óptima.

.

**Grafica comparando Training y Test**

Repetimos el proceso aplicando stratiffiedshufflesplit al dataset para que el dataset de training no sea siempre el mismo y predecimos la accuracy sobre los datos de test y sobre los de training.

#Repito el proceso 20 veces estratificando el dataset para obtener la curva

misss = StratifiedShuffleSplit(20,0.33)

index = 0

Res = []

Restrain = []

Ytrain = []

Ytest = []

for train\_index, test\_index in misss.split(X,Y):

Xtrain = X[train\_index,:]

Xtest = X[test\_index,:]

Ytrain = Y[train\_index]

Ytest = Y[test\_index]

Restrain = cross\_val\_score(miKvecinos, Xtrain, Ytrain,cv=20)

Restest = cross\_val\_score(miKvecinos, Xtest, Ytest,cv=20)

index = index +1

#Grafica de las accuracys para Test y Train

arrayx= range(1,21)

plt.plot(arrayx,Restrain, 'r--',label='Accuracy Train')

plt.plot(arrayx,Restest, 'g--',label='Accuracy Test')

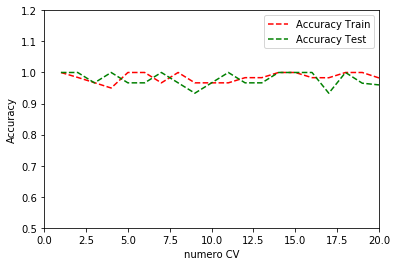
plt.axis([0, 20, 0, 1.2])

plt.legend(loc='up right')

plt.xlabel('numero CV')

plt.ylabel('Accuracy')

plt.show()



Podemos observar que la accuracy para predecir el Test y el Train es bastante no depende del número de crosvalidaciones y es bastante optimo, aunque el número de cros validaciones sea bajo. Esto es porque el dataset tiene una buena distribución.

#### KNN3 Aplicación del método GridSearch K-vecinos para obtener distintas accuracys cuando varían los parámetros de modelo de kvecinos. Se crea una rejilla con los resultados de cada iteración

from sklearn.datasets import load\_digits

from sklearn.neighbors import KNeighborsClassifier

import numpy as np

misdatos= load\_digits() **#cargo los datos**

X=misdatos.data **#creo la matriz X**

Y=misdatos.target **# vector Y de las etiquetas**

miKvecinos=KNeighborsClassifier

from sklearn.model\_selection import GridSearchCV

**#importa el método GridSearchCV hace una busqueda por rejilla.**

mi\_param\_grid={'n\_neighbors':[3,5,7,9,11,13,15],'weights':['uniform','distance']} **#número de vecinos: valores a,b,c,d de la rejilla**

migscv=GridSearchCV(miKvecinos,mi\_param\_grid,cv=10,verbose=2)

**#verbose hace que salga por pantalla lo que va haciendo**

migscv.fit(X,Y)

print("migcsc"+ str(migscv))

**#Fitting 10 folds for each of 14 candidates, totalling 140 fits 7 (a,,b,c,d,...) (3,5,7,9...\* 2 (alfa,beta) uniform distanre**

**#migscv.best\_estimator\_**

miMejorKvecinos=migscv.best\_estimator\_ **#se escoge la mejor opción de calibración**

miMejorKvecinos.fit(X,Y) **#el modelo aprende**

print("mimejor" +str(miMejorKvecinos)) **#se imprime el resultado**

**Conclusión**:

***Resultado***

***Después de hacer la rejilla el mejor modelo es:***

print (migscv.best\_estimator\_)

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=3, p=2,

weights='distance')

***Y sus medias y varianzas son:***

print (migscv.grid\_scores\_)

[mean: 0.97774, std: 0.01594, params: {'n\_neighbors': 3, 'weights': 'uniform'},

mean: 0.97830, std: 0.01605, params: {'n\_neighbors': 3, 'weights': 'distance'},

mean: 0.97385, std: 0.01655, params: {'n\_neighbors': 5, 'weights': 'uniform'},

mean: 0.97385, std: 0.01655, params: {'n\_neighbors': 5, 'weights': 'distance'},

mean: 0.96995, std: 0.01919, params: {'n\_neighbors': 7, 'weights': 'uniform'},

mean: 0.96995, std: 0.01920, params: {'n\_neighbors': 7, 'weights': 'distance'},

mean: 0.96717, std: 0.02008, params: {'n\_neighbors': 9, 'weights': 'uniform'},

mean: 0.96995, std: 0.02108, params: {'n\_neighbors': 9, 'weights': 'distance'},

mean: 0.96550, std: 0.02123, params: {'n\_neighbors': 11, 'weights': 'uniform'},

mean: 0.96772, std: 0.01876, params: {'n\_neighbors': 11, 'weights': 'distance'},

mean: 0.96494, std: 0.02244, params: {'n\_neighbors': 13, 'weights': 'uniform'},

mean: 0.96550, std: 0.02227, params: {'n\_neighbors': 13, 'weights': 'distance'},

mean: 0.96439, std: 0.02147, params: {'n\_neighbors': 15, 'weights': 'uniform'},

mean: 0.96605, std: 0.02236, params: {'n\_neighbors': 15, 'weights': 'distance'}]

Este método te facilita saber cuál es el mejor ajuste para el modelo que estás utilizando, sin tener que ir probando uno a uno.

Como parámetros pasamos solo los impares para que sea equidistante.

En este caso el mejor método de kvecinos es con los parámetros 3 vecinos y el peso uniforme.

**Grafica con la comparativa de las accuracys para Train y Test**

#plotear grafica de scores train/test

import matplotlib.pyplot as plt

import numpy as np

from sklearn.svm import SVC

from sklearn.model\_selection import validation\_curve

datacancer = load\_breast\_cancer()

X= datacancer.data

Y= datacancer.target

estimator=KNeighborsClassifier()

param\_range = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

train\_scores, test\_scores = validation\_curve(

estimator, X, Y, param\_name="n\_neighbors", param\_range=param\_range,

cv=2, scoring="accuracy", n\_jobs=1)

train\_scores\_mean = np.mean(train\_scores, axis=1)

train\_scores\_std = np.std(train\_scores, axis=1)

test\_scores\_mean = np.mean(test\_scores, axis=1)

test\_scores\_std = np.std(test\_scores, axis=1)

plt.title("Validation Curve with GS")

plt.xlabel("Kvecinos")

plt.ylabel("Score")

plt.ylim(0.0, 1.1)

plt.xlim(1, 15)

lw = 2

plt.plot(param\_range, train\_scores\_mean, label="Training score",

color="darkorange", lw=lw)

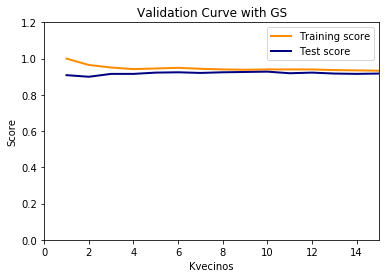
plt.plot(param\_range, test\_scores\_mean, label="Test score",

color="navy", lw=lw)

plt.legend(loc="best")

plt.axis([0,15, 0, 1.2])

plt.show()



En la gráfica se observa que cuanto mayor es el parámetro kvecinos mas similares son las accuracys obtenidas con Train y Test, lo que significa que la predicción es mas ajustada y es un buen modelo para este dataset.

A partir de 3 vecinos la acuracy empieza a ser muy similar para los datos de Test y los de Train siendo un valor bastante óptimo.

kvecinos genera es un buen método para este dataset.

#### KNN4

Se eliminan características del dataset con la función PCA para poder ilustrar el dataset en dos dimensiones

from sklearn.datasets import load\_iris

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import StratifiedShuffleSplit

from matplotlib import pyplot as plt

from sklearn.decomposition import PCA

import numpy as np

misdatos=load\_iris()

X=misdatos.data

Y=misdatos.target

miPCA=PCA(n\_components=2)

**#eliminamos columnas de “features” para poder pintar una proyección**

**#por componentes principales pongo 2 que es igual que el num de columnas transformadas**

X\_PCA=miPCA.fit\_transform(X)

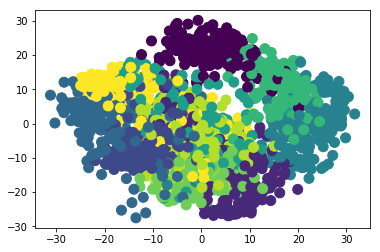
**#has proyectado todo manteniendo la máxima varianza**

**#eliminas la varianza cero porque no te aporta nada nuevo**

plt. scatter(X\_PCA[:,0],X\_PCA[:,1],s=100,c=Y)

plt.show()

**#Vamos a pintar las etiquetas con s=100, c=Y**



**Grafica de Covarianza**

#grafica covarianza

from sklearn.decomposition import PCA

covar\_matrix = PCA(n\_components = 30) #we have 30 features

covar\_matrix.fit(X)

variance = covar\_matrix.explained\_variance\_ratio\_ #calculate variance ratios

var=np.cumsum(np.round(covar\_matrix.explained\_variance\_ratio\_, decimals=3)\*100)

print(var)

plt.ylabel('% Variance Explained')

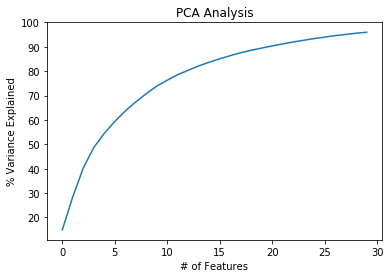
plt.xlabel('# of Features')

plt.title('PCA Analysis')

#plt.ylim(30,100.5)

plt.style.context('seaborn-whitegrid')

plt.plot(var)



Si cogemos solo dos features para poder plotear la proyección estamos perdiendo mucha información ya que la covarianza es muy baja, alrededor de 20. La ilustración de este dataset con dos variables no seria significativa.

### Configuramos el script de validación automatizada para que la función GridSearchCV() utilice validación leave-one-out en lugar de k-fold.

Usamos el Gridsearch con cros validación = LeaveOneOut y con crosvalidacion = 10 para así comparar los resultados de los dos métodos. Usamos kvecinos impares para evitar empates.

**CODIGO**

from sklearn.datasets import load\_digits

from sklearn.neighbors import KNeighborsClassifier

import numpy as np

from sklearn.model\_selection import LeaveOneOut

from sklearn.metrics import accuracy\_score

from sklearn.model\_selection import StratifiedShuffleSplit

datacancer = load\_digits()

X= datacancer.data

Y= datacancer.target

############################### LEAVE ONE OUT ####################

#Dividimos el dataset en Train(2/3) y test(1/3)

misss = StratifiedShuffleSplit(1, 0.33)

index=0

for train\_index, test\_index in misss.split(X, Y):

Xtrain=X[train\_index,:]

Xtest=X[test\_index,:]

Ytrain=Y[train\_index]

Ytest=Y[test\_index]

index=index+1

#Aplicamos el metodo Grid Search para el elegir el mejor modelo para nuestro data set

#Con CV de leaveOneOut, que es equivalente a hacer una CV con tanatas k-folds como ejemplos

#tiene nuestro dataset, en este caso 569 ejemplos.

miKvecinos=KNeighborsClassifier()

from sklearn.model\_selection import GridSearchCV

mi\_param\_grid={'n\_neighbors':[3,4,5,6,7,8,9,10,11], 'weights':['uniform','distance']}

migscv=GridSearchCV(miKvecinos,mi\_param\_grid,cv=LeaveOneOut(),verbose=2)

migscv.fit(Xtrain,Ytrain)

miMejorKvecinosLOO=migscv.best\_estimator\_

miMejorKvecinosLOO.fit(Xtrain,Ytrain)

Ypred = miMejorKvecinosLOO.predict(Xtest)

accuracy\_score(Ytest, Ypred

#Repito el proceso de stratificacion 20 veces, entreno con mi mejor modelo de LOO

# y saco las accuracys de las 2¶0 iteraciones con Xtest y con Xtrain

misss = StratifiedShuffleSplit(20,0.33)

index = 0

ResLOO = []

ResLOOtrain = []

for train\_index, test\_index in misss.split(X,Y):

   Xtrain = X[train\_index,:]

   Xtest = X[test\_index,:]

   Ytrain = Y[train\_index]

   Ytest = Y[test\_index]

   miMejorKvecinosLOO.fit(Xtrain,Ytrain)

   Ypred = miMejorKvecinosLOO.predict(Xtest)

   Ypredtrain = miMejorKvecinosLOO.predict(Xtrain)

   print (accuracy\_score(Ytest, Ypred))

   ResLOO.append(accuracy\_score(Ytest, Ypred))

   ResLOOtrain.append(accuracy\_score(Ytrain, Ypredtrain))

   index = index +1

print ("Accuracy de LOO: " + str(ResLOO))

print ("Media LOO:" + str(np.mean(ResLOO)))

print ("DevLOO:" + str(np.std(ResLOO)))

print ("Accuracy de LOOtrain: " + str(ResLOOtrain))

print ("Media LOOtrain:" + str(np.mean(ResLOOtrain)))

print ("DevLOOtrain:" + str(np.std(ResLOOtrain)))

#Grafica con la comparacion de accuracy para Ytest e Ytrain con el mejor modelo

#Y con LOO

plt.plot([3, 3], [np.mean(ResLOO),np.mean(ResLOOtrain) ])

plt.axis([0, 10, 0.9, 1])

plt.show()

arrayx= range(1,21)

plt.plot(arrayx,ResLOO, 'r--', arrayx,ResLOOtrain, 'g--')

plt.axis([0, 20, 0.8, 1])

plt.xlabel('numero iteracion')

plt.ylabel('Accuracy')

plt.show()

############### CROSS VALIDATION ######################33

#Dividimos el dataset en Train(2/3) y test(1/3)

misss = StratifiedShuffleSplit(1, 0.33)

#GridSearch Con 10 CV

migscv=GridSearchCV(miKvecinos,mi\_param\_grid,cv=10,verbose=2)

migscv.fit(Xtrain,Ytrain)

#Mejor estimador

print (migscv.best\_estimator\_)

#Media y varianza para todos los modelos

print (migscv.grid\_scores\_)

miMejorKvecinosCV=migscv.best\_estimator\_

miMejorKvecinosCV.fit(Xtrain,Ytrain)

Ypred = miMejorKvecinosCV.predict(Xtest)

accuracy\_score(Ytest, Ypred)

#Repito el proceso de stratificacion 20 veces con mi mejor modelo de CV

index = 0

ResCV = []

ResCVtrain = []

misss = StratifiedShuffleSplit(20,0.33)

for train\_index, test\_index in misss.split(X,Y):

Xtrain = X[train\_index,:]

Xtest = X[test\_index,:]

Ytrain = Y[train\_index]

Ytest = Y[test\_index]

miMejorKvecinosCV.fit(Xtrain,Ytrain)

Ypred = miMejorKvecinosCV.predict(Xtest)

Ypredtrain = miMejorKvecinosCV.predict(Xtrain)

print (accuracy\_score(Ytest, Ypred))

print (accuracy\_score(Ytrain, Ypredtrain))

ResCV.append(accuracy\_score(Ytest, Ypred))

ResCVtrain.append(accuracy\_score(Ytrain, Ypredtrain))

index = index +1

#Resultados

print ("Accuracy de CV: " + str(ResCV))

print ("Media:" + str(np.mean(ResCV)))

print ("Dev:" + str(np.std(ResCV)))

print ("Accuracy de CV train: " + str(ResCVtrain))

print ("Media:" + str(np.mean(ResCVtrain)))

print ("Dev:" + str(np.std(ResCVtrain)))

#Pintar grafica con las curvas de la accuracy para train y para test

import matplotlib.pyplot as plt

plt.plot([11, 11], [np.mean(ResCV),np.mean(ResCVtrain) ])

plt.plot([3, 3], [np.mean(ResLOO),np.mean(ResLOOtrain) ])

plt.axis([0, 12, 0.8, 1])

plt.show()

**RESULTADO**:

**Con leave one out**

Mi mejor modelo

print (migscv.best\_estimator\_)

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=4, p=2,

weights='distance')

Scores de todos los modelos:

print (migscv.grid\_scores\_)

[mean: 0.98504, std: 0.12140, params: {'n\_neighbors': 3, 'weights': 'uniform'},

mean: 0.98587, std: 0.11803, params: {'n\_neighbors': 3, 'weights': 'distance'},

mean: 0.98254, std: 0.13096, params: {'n\_neighbors': 4, 'weights': 'uniform'},

mean: 0.98670, std: 0.11456, params: {'n\_neighbors': 4, 'weights': 'distance'},

mean: 0.98504, std: 0.12140, params: {'n\_neighbors': 5, 'weights': 'uniform'},

mean: 0.98587, std: 0.11803, params: {'n\_neighbors': 5, 'weights': 'distance'},

mean: 0.98005, std: 0.13983, params: {'n\_neighbors': 6, 'weights': 'uniform'},

mean: 0.98670, std: 0.11456, params: {'n\_neighbors': 6, 'weights': 'distance'},

mean: 0.98005, std: 0.13983, params: {'n\_neighbors': 7, 'weights': 'uniform'},

mean: 0.98337, std: 0.12786, params: {'n\_neighbors': 7, 'weights': 'distance'},

mean: 0.98005, std: 0.13983, params: {'n\_neighbors': 8, 'weights': 'uniform'},

mean: 0.98254, std: 0.13096, params: {'n\_neighbors': 8, 'weights': 'distance'},

mean: 0.97922, std: 0.14265, params: {'n\_neighbors': 9, 'weights': 'uniform'},

mean: 0.98088, std: 0.13694, params: {'n\_neighbors': 9, 'weights': 'distance'},

mean: 0.97672, std: 0.15078, params: {'n\_neighbors': 10, 'weights': 'uniform'},

mean: 0.98088, std: 0.13694, params: {'n\_neighbors': 10, 'weights': 'distance'},

mean: 0.97672, std: 0.15078, params: {'n\_neighbors': 11, 'weights': 'uniform'},

mean: 0.97922, std: 0.14265, params: {'n\_neighbors': 11, 'weights': 'distance'}]

**Score del mejor modelo:**

0.99158249158249157

**Aplicamos el mejor modelo a 20 muestras distintas de Xtrain y de Xtest  con el shuffledSplit y obtenemos los scores:**

Resultados prediciendo con Ytest

**Accuracys**

Accuracy de LOO: [[0.98989898989898994, 0.98148148148148151, 0.98653198653198648, 0.98484848484848486, 0.99158249158249157, 0.97643097643097643, 0.98821548821548821, 0.98316498316498313, 0.98989898989898994, 0.98821548821548821, 0.98148148148148151, 0.98821548821548821, 0.9932659932659933, 0.98316498316498313, 0.98653198653198648, 0.97643097643097643, 0.98653198653198648, 0.99158249158249157, 0.97306397306397308, 0.98316498316498313]]

**Media**

Media LOO: 0.985185185185

**Desviacion**

DevLOO: 0.00531304169632

Resultados prediciendo con Ytrain

**Accuracys**

Accuracy de LOOtrain: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

**Media**

Media LOOtrain: 1.0

**Desviacion**

DevLOOtrain:0.0

**Con CV**

Mi mejor modelo

print (migscv.best\_estimator\_)

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=3, p=2,

weights='uniform')

Scores de todos los modelos:

print (migscv.grid\_scores\_)

[mean: 0.98753, std: 0.01006, params: {'n\_neighbors': 3, 'weights': 'uniform'}, mean: 0.98753, std: 0.01006, params: {'n\_neighbors': 3, 'weights': 'distance'}, mean: 0.98005, std: 0.01287, params: {'n\_neighbors': 4, 'weights': 'uniform'}, mean: 0.98670, std: 0.01056, params: {'n\_neighbors': 4, 'weights': 'distance'}, mean: 0.98504, std: 0.01271, params: {'n\_neighbors': 5, 'weights': 'uniform'}, mean: 0.98587, std: 0.01110, params: {'n\_neighbors': 5, 'weights': 'distance'}, mean: 0.97839, std: 0.01449, params: {'n\_neighbors': 6, 'weights': 'uniform'}, mean: 0.98421, std: 0.01235, params: {'n\_neighbors': 6, 'weights': 'distance'}, mean: 0.97672, std: 0.01423, params: {'n\_neighbors': 7, 'weights': 'uniform'}, mean: 0.97839, std: 0.01441, params: {'n\_neighbors': 7, 'weights': 'distance'}, mean: 0.97589, std: 0.01084, params: {'n\_neighbors': 8, 'weights': 'uniform'}, mean: 0.97839, std: 0.01347, params: {'n\_neighbors': 8, 'weights': 'distance'}, mean: 0.97506, std: 0.01292, params: {'n\_neighbors': 9, 'weights': 'uniform'}, mean: 0.97756, std: 0.01351, params: {'n\_neighbors': 9, 'weights': 'distance'}, mean: 0.97340, std: 0.01049, params: {'n\_neighbors': 10, 'weights': 'uniform'}, mean: 0.97756, std: 0.01340, params: {'n\_neighbors': 10, 'weights': 'distance'}, mean: 0.97257, std: 0.01186, params: {'n\_neighbors': 11, 'weights': 'uniform'}, mean: 0.97839, std: 0.01002, params: {'n\_neighbors': 11, 'weights': 'distance'}]

**Score del mejor modelo:**

0.98653198653198648

**Aplicamos el mejor modelo a 20 muestras distintas de Xtrain y de Xtest  con el shuffledSplit y obtenemos los scores:**

Resultados prediciendo con Ytest

**Accuracys**

[[0.98821548821548821, 0.98653198653198648, 0.99158249158249157, 0.98316498316498313, 0.98484848484848486, 0.98989898989898994, 0.98316498316498313, 0.98653198653198648, 0.98484848484848486, 0.97979797979797978, 0.98821548821548821, 0.98821548821548821, 0.99158249158249157, 0.98148148148148151, 0.98989898989898994, 0.97306397306397308, 0.98989898989898994, 0.98316498316498313, 0.98653198653198648, 0.99158249158249157]]

**Media**

Media: 0.986111111111

**Desviacion**

Dev: 0.00451338606682

Resultados prediciendo con Ytrain

**Accuracys**

0.99501246882793015, 0.9900249376558603, 0.9908561928512053, 0.99418121363258516, 0.99251870324189528, 0.98919368246051542, 0.99334995843724028, 0.99251870324189528, 0.99251870324189528, 0.99584372402327515, 0.99251870324189528, 0.99251870324189528, 0.99251870324189528, 0.99334995843724028, 0.9908561928512053, 0.99418121363258516, 0.99418121363258516, 0.99168744804655029, 0.99334995843724028, 0.99501246882793015]

**Media**

Media: 0.99280964256

**Desviacion**

Dev: 0.00166821548133

**GRAFICAS**

**CODIGO**

#Grafica con la comparacion de accuracy para Ytest e Ytrain con el mejor modelo de LOO

plt.plot([3, 3], [np.mean(ResLOO),np.mean(ResLOOtrain) ])

plt.axis([0, 10, 0.9, 1])

plt.show()

arrayx= range(1,21)

plt.plot(arrayx,ResLOO, 'r--', label = 'Test')

plt.plot(arrayx,ResLOOtrain, 'g--', label='Train')

plt.axis([0, 20, 0.8, 1.1])

plt.legend(loc='downer right')

plt.xlabel('numero iteracion')

plt.ylabel('Accuracy')

plt.show()

#Grafica con la comparacion de accuracy para Ytest e Ytrain con el mejor modelo de CV

plt.plot([3, 3], [np.mean(ResCV),np.mean(ResCVtrain) ])

plt.axis([0, 10, 0.9, 1])

plt.show()

arrayx= range(1,21)

plt.plot(arrayx,ResCV, 'r--', label = 'Test')

plt.plot( arrayx,ResCVtrain, 'g--', label = 'Train')

plt.axis([0, 20, 0.8, 1.1])

plt.legend(loc='downer right')

plt.xlabel('numero iteracion')

plt.ylabel('Accuracy')

plt.show()

#Pintar grafica con las curvas de la accuracy para train y para test

import matplotlib.pyplot as plt

plt.plot([11, 11], [np.mean(ResCV),np.mean(ResCVtrain)] ,label='CV' )

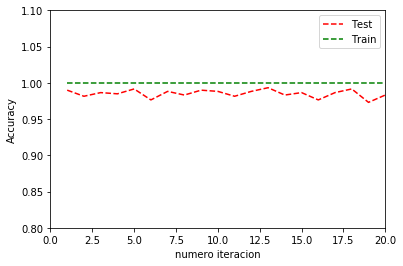
plt.plot([3, 3], [np.mean(ResLOO),np.mean(ResLOOtrain) ], label = 'LeaveOneOut')

plt.axis([0, 12, 0.9, 1])

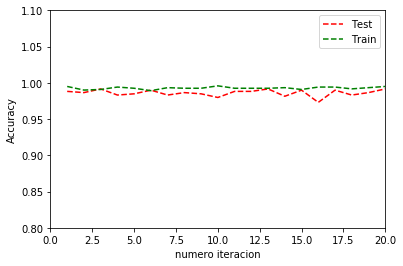
plt.legend(loc='downer right')

plt.show()

Gráfica con las accuracys obtenidas al aplicar el mejor modelo de LeaveOneOut a 20 dataset shuffleados con kvecinos =3



Gráfica con las accuracys obtenidas al aplicar el mejor modelo de CV a 20 dataset shuffleados con kvecinos =3



Comparando estas dos graficas: El método de LOO para los datos de training tiene una accuracy de 1 y desviación de cero.

En este método el subconjunto de training tiene una cardinalidad de (n-samples -1), por lo que al predecir luego el target de training la accuracy es siempre 1, siempre acierta ya que está muy bien entrenado.

Con el método de CV la accuracy de LOO y de CV es bastante similar, el modelo ajusta bien.

Conclusion: cuanto mayor es el subconjunto de datos de training mayor es la accuracy de training.

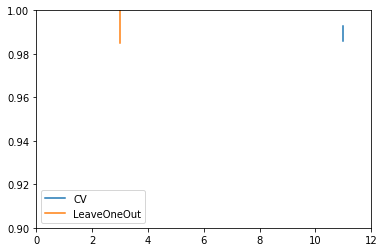
Gráfica con la comparación de las medias de las accuracys obtenidas al aplicar el mejor modelo de LeaveOneOut a 20 dataset shuffleados con kvecinos = 3.

￼￼

**Gráfica que compara las medias de las accuracys obtenidas para los dos métodos**

Punto 11 = CV

Punto 3 = Leave One Out



**CONCLUSIONES**

- Para este dataset el método LeaveOneOut tienes un coste computacional muy elevado, tarda más de media hora en hacer el GridSearch ya que son muchas iteraciones.

La validación leave-one-out recorre todo el dataset haciendo cros validación de un ejemplo con todos los demás, es como hacer un kfold con tantas particiones como ejemplos tienes la muestra. El gasto computacional al ejecutarlo es más elevado que en el Kfold con 10 folds ya que el numero de iteraciones es mucho más elevado con LeaveOneOut.

* En los dos casos la accuracy se mantiene bastante constante por lo que ambos modelos son estables y consistentes. Es decir, el mejor kvecinos en cada forma de crosvalidar da un buen resultado sobre la muestra.
* Comparando los dos métodos el de CV es ams oprtimo ya que tiene menor gasto computacional y la accuracy es mejor.

### ¿Es necesario estratificar el proceso de cross-validación analizando la distribución de muestras por clase?.

Estratificar es el proceso de subdividir el dataset en grupos homogéneos antes de coger las muestras. El proceso consiste en ordenar las muestras de manera que la parte del dataset seleccionada para entrenar tiene una representación homogénea de todas las muestras o que, si estas no están balanceadas y se quiere dar relevancia a las minoritarias, se incluyan en mayor medida.

import matplotlib.pyplot as plt

print('unos de las etiquetas' + str(sum(Y==0)))

Y0=sum(Y==0)

Y1=sum(Y==1)

Y2=sum(Y==2)

Y3=sum(Y==3)

Y4=sum(Y==4)

Y5=sum(Y==5)

Y6=sum(Y==6)

Y7=sum(Y==7)

Y8=sum(Y==8)

Y9=sum(Y==9)

labels = '0','1', '2', '3', '4','5','6','7','8','9'

sizes = [Y0, Y1, Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9]

explode = (0, 0.1, 0, 0,0,0,0,0,0,0)

fig1, ax1 = plt.subplots()

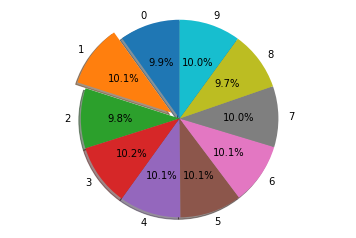
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',

shadow=True, startangle=90)

ax1.axis('equal')

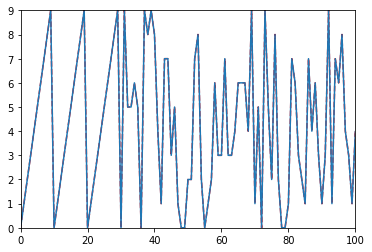
plt.show()

En el caso del dataset de digits se puede ver que el numero de labels de cada tipo es similar:



### 

Si ploteamos las 100 primeras labels para ver su distribución obtenemos esta gráfica:



Parece que las labels están bastante distribuidas dentro del dataset.

### Introducimos en el proceso de cross-validación el ajuste de los pesos de la métrica de distancia entre muestras de acuerdo al parámetro “weights” del modelo en scikit-learn:

from sklearn.datasets import load\_digits

from sklearn.neighbors import KNeighborsClassifier

import numpy as np

from sklearn.metrics import accuracy\_score

data = load\_digits()

X= data.data

Y= data.target

#Dividimos el dataset en Train(2/3) y test(1/3)

misss = StratifiedShuffleSplit(1, 0.33)

for trai\_index, test\_index in misss.split(X,Y):

   Xtrain = X[train\_index,:]

   Xtest = X[test\_index,:]

   Ytrain = Y[train\_index]

   Ytest = Y[test\_index]

   index = index +1

#comparo con GridSearch uniform/distance

mi\_param\_grid={'n\_neighbors':[3], 'weights':['uniform','distance']}

migscv=GridSearchCV(miKvecinos,mi\_param\_grid,cv=10,verbose=2)

migscv.fit(Xtrain,Ytrain)

#Mejor estimador

print (migscv.best\_estimator\_)

#Media y varianza para todos los modelos

print (migscv.grid\_scores\_)

#Repito el proceso de stratificacion 20 veces

# y saco las accuracys de las 20 iteraciones para peso uniforme y distancia

misss = StratifiedShuffleSplit(20,0.33)

index = 0

ASU = []

ASD = []

RESULTADO

best\_estimator\_

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=3, p=2,

weights='distance')

Weigth = uniform

Accuracy de uniform: [0.99158249158249157, 0.98148148148148151, 0.98821548821548821, 0.98821548821548821, 0.98989898989898994, 0.98316498316498313, 0.98316498316498313, 0.98484848484848486, 0.98148148148148151, 0.98316498316498313, 0.99158249158249157, 0.99158249158249157, 0.99158249158249157, 0.98653198653198648, 0.97643097643097643, 0.98989898989898994, 0.98653198653198648, 0.97643097643097643, 0.98653198653198648, 0.98989898989898994

Media uniform: 0.986111111111

Dev uniform: 0.00463727559624

for train\_index, test\_index in misss.split(X,Y):

Xtrain = X[train\_index,:]

Xtest = X[test\_index,:]

Ytrain = Y[train\_index]

Ytest = Y[test\_index]

miKvecinos = KNeighborsClassifier(n\_neighbors=3, weights = 'uniform')

miKvecinos.fit(Xtrain,Ytrain)

Ypred = miKvecinos.predict(Xtest)

ASU.append(accuracy\_score(Ytest, Ypred))

miKvecinos = KNeighborsClassifier(n\_neighbors=3, weights = 'distance')

miKvecinos.fit(Xtrain,Ytrain)

Ypred = miKvecinos.predict(Xtest)

accuracy\_score(Ytest, Ypred)

ASD.append(accuracy\_score(Ytest, Ypred))

index = index +1

print (ASU)

print (ASD)

print ("Accuracy de unifrom: " + str(ASU))

print ("Media uniform:" + str(np.mean(ASU)))

print ("Dev uniform:" + str(np.std(ASU)))

print ("Accuracy de distance: " + str(ASD))

print ("Media distance:" + str(np.mean(ASD)))

print ("Dev distance:" + str(np.std(ASD))

RESULTADO

Weigth = distance

Accuracy de distance: [0.99158249158249157, 0.98148148148148151, 0.98821548821548821, 0.98653198653198648, 0.99158249158249157, 0.98316498316498313, 0.98484848484848486, 0.98484848484848486, 0.98148148148148151, 0.98484848484848486, 0.99494949494949492, 0.99158249158249157, 0.9932659932659933, 0.98653198653198648, 0.97643097643097643, 0.98989898989898994, 0.98653198653198648, 0.97811447811447816, 0.98484848484848486, 0.98989898989898994]

Media distance: 0.986531986532

Dev distance: 0.00482081517282

**CONCLUSION**

Al hacer el Grid search el mejor estimador con 3 kvecinos es con el peso de distancia.

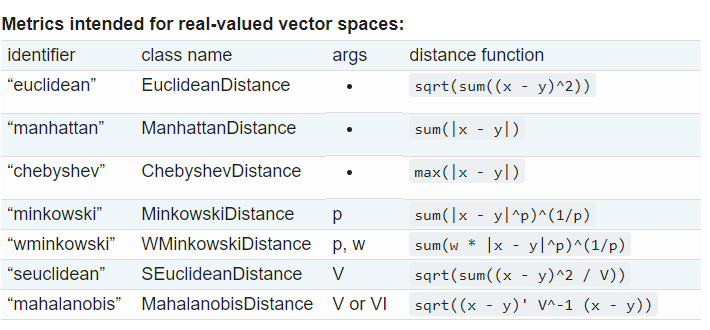
Repetimos el proceso 20 veces con el dataset shuffleado para obtener accuracys con distintos subconjuntos del dataset para los dos tipos de peso.

La media de la acuracy es muy buena, el método modela bien el dataset.

Esto significa que al darle mayor influencia a los puntos cercanos el resultado es óptimo, es decir que los puntos mas cercanos a uno dado tienen mas influencia que el resto.

### Introducimos el tipo de métrica de distancia (parámetro “metric”) dentro del proceso de validación cruzada.

Los tipos de métricas son los siguientes:



Comparamos las cuatro primeras.

#Valid metrics are ['euclidean', 'l2', 'l1', 'manhattan', 'cityblock',

#                   'braycurtis', 'canberra', 'chebyshev', 'correlation',

#                   'cosine', 'dice', 'hamming', 'jaccard', 'kulsinski',

#                   'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto',

#                   'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath',

#                   'sqeuclidean', 'yule', 'wminkowski'], or 'precomputed', or a

#                   callable

from sklearn.datasets import load\_digits

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import StratifiedShuffleSplit

from sklearn.model\_selection import GridSearchCV

from sklearn.model\_selection import cross\_val\_score

import numpy as np

from sklearn.metrics import accuracy\_score

data = load\_digits()

X= data.data

Y= data.target

#dividimos el Dataset en Xtrain y Xtest

misss = StratifiedShuffleSplit(1, 0.33)

miKvecinos=KNeighborsClassifier()

#aplicamos el metodo GridSearch para hayar la mejor metrica para nuestro modelo con K-vecinos = 5

mi\_param\_grid={'n\_neighbors':[5], 'metric' : ["euclidean", "manhattan", "chebyshev", "minkowski", "l1", "l2", "cityblock", 'hamming', 'jaccard', 'kulsinski'] }

migscv=GridSearchCV(miKvecinos,mi\_param\_grid,cv=2,verbose=2)

migscv.fit(X,Y)

print (migscv.best\_estimator\_)

print (migscv.cv\_results\_)

miMejorKvecinos=migscv.best\_estimator\_

miMejorKvecinos.fit(Xtrain,Ytrain)

Ypred = miMejorKvecinos.predict(Xtest)

accuracy\_score(Ytest, Ypred)

print (migscv.grid\_scores\_)

print (migscv.best\_score\_)

#Repito el proceso de stratificacion 20 veces, entreno con mi mejor con la distancia manhattan

# y saco las accuracys de las 2¶0 iteraciones con Xtest y con Xtrain

misss = StratifiedShuffleSplit(20,0.33)

index = 0

Res = []

Restrain = []

for train\_index, test\_index in misss.split(X,Y):

   Xtrain = X[train\_index,:]

   Xtest = X[test\_index,:]

   Ytrain = Y[train\_index]

   Ytest = Y[test\_index]

   miMejorKvecinos.fit(Xtrain,Ytrain)

   Ypred = miMejorKvecinos.predict(Xtest)

   Ypredtrain = miMejorKvecinos.predict(Xtrain)

   print (accuracy\_score(Ytest, Ypred))

   Res.append(accuracy\_score(Ytest, Ypred))

   Restrain.append(accuracy\_score(Ytrain, Ypredtrain))

   index = index +1

#Accuracis prediciendo con Ytest

print (Res)

#Accuracis prediciendo con Ytrain

print (Restrain)

Mi mejor estimador

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='euclidean',

metric\_params=None, n\_jobs=1, n\_neighbors=5, p=2,

weights='uniform')

Grid scores

print (migscv.grid\_scores\_)

[mean: 0.95270, std: 0.00154, params: {'n\_neighbors': 5, 'metric': 'euclidean'},

mean: 0.94157, std: 0.00373, params: {'n\_neighbors': 5, 'metric': 'manhattan'},

mean: 0.93044, std: 0.01483, params: {'n\_neighbors': 5, 'metric': 'chebyshev'},

mean: 0.95270, std: 0.00154, params: {'n\_neighbors': 5, 'metric': 'minkowski'},

mean: 0.94157, std: 0.00373, params: {'n\_neighbors': 5, 'metric': 'l1'},

mean: 0.95270, std: 0.00154, params: {'n\_neighbors': 5, 'metric': 'l2'},

mean: 0.94157, std: 0.00373, params: {'n\_neighbors': 5, 'metric': 'cityblock'},

mean: 0.79633, std: 0.02058, params: {'n\_neighbors': 5, 'metric': 'hamming'},

mean: 0.84474, std: 0.00680, params: {'n\_neighbors': 5, 'metric': 'jaccard'},

mean: 0.81024, std: 0.00442, params: {'n\_neighbors': 5, 'metric': 'kulsinski'}]

**best score**

0.952698942682

Entrenamos con el mejor 20 versiones stratificadas del data set.

Datos de accuracy prediciendo con Ytest

Accuracy : [0.98484848484848486, 0.9932659932659933, 0.98316498316498313, 0.97979797979797978, 0.97643097643097643, 0.9932659932659933, 0.98484848484848486, 0.98821548821548821, 0.98989898989898994, 0.98989898989898994, 0.98989898989898994, 0.98484848484848486, 0.97979797979797978, 0.98653198653198648, 0.98148148148148151, 0.98316498316498313, 0.98148148148148151, 0.97643097643097643, 0.98821548821548821, 0.98316498316498313]

Media: 0.984932659933

Dev: 00483475773763

print ("Accuracy : " + str(Res))

print ("Media:" + str(np.mean(Res)))

print ("Dev:" + str(np.std(Res)))

print ("Accuracy de train: " + str(Restrain))

print ("Media train:" + str(np.mean(Restrain)))

print ("Devtrain:" + str(np.std(Restrain)))

plt.plot([5, 5], [np.mean(Res),np.mean(Restrain) ])

plt.axis([0, 10, 0.8, 1])

plt.show()

.

Datos de accuracy prediciendo con Ytrain

[0.99251870324189528, 0.9900249376558603, 0.9900249376558603, 0.9900249376558603, 0.9900249376558603, 0.98836242726517043, 0.98919368246051542, 0.9908561928512053, 0.9908561928512053, 0.9900249376558603, 0.9908561928512053, 0.99251870324189528, 0.99168744804655029, 0.98836242726517043, 0.9900249376558603, 0.9900249376558603, 0.9900249376558603, 0.9900249376558603, 0.98586866167913545, 0.99251870324189528]

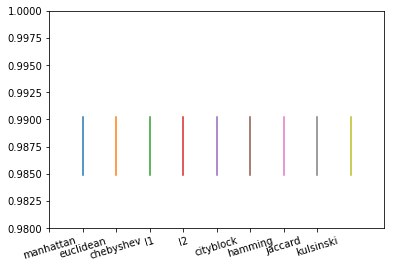
Media train:0.990191188695

Devtrain:0.0015237159418

￼

Gráfica de las medias de las accuracys de train y test con k-vecinos = 5, métricas manhattan, euclidean, minkowski, l1, l2, cityblock, haming, jaccard, kulsinski

y estratificando el dataset 20 veces:



**CONCLUSIONES**

El mejor modelo elegido para 5 k-vecinos es la métrica euclídea, la que mide la distancia real con la sima de los cuadrados de las coordenadas de los puntos.

. Al filtrar este modelo con 20 datasets estratificados y sacar las acuracis vemos que el modelo elegido es consistente ya que la media y desviación son óptimas.

Las accuracys para todas las métricas son muy buenas y hay poca diferencia entre una u otra, por lo que los target del dataset están concentrados.