Aprendizaje Automático (2017-2018)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Proyecto Final:
Breast Cancer Wisconsin (Diagnostic)

Francisco Solano López Rodríguez
Simón López Vico

6 de junio de 2018

Índice

1. Descripción del problema.

Comencemos este proyecto comentando del problema sobre el que vamos a trabajar y el conjunto de datos que queremos ajustar.

El problema a realizar se basa en el diagnóstico de cáncer de mama a través del ajuste de la información proporcionada por una base de datos.

Las características de los datos proporcionados para el ajuste han sido calculadas a partir de imágenes digitalizadas de un aspirado con aguja fina (AAF) de una masa mamaria. Dichos atributos describen las propiedades de los núcleos celulares presentes en la imagen.

Las características relevantes ya han sido seleccionadas mediante una búsqueda exhaustiva en el espacio de 1-4 características y 1-3 planos de separación.

Los atributos de nuestro conjunto de datos serán:

1) Número ID.

2) Diagnóstico (M=maligno, B=benigno).

3-32) Se calculan diez características de valor real para cada núcleo celular:

- Radio (media de las distancias del centro a los puntos del perímetro).

- Textura (desviación estándar de los valores de la escala de grises).
- Perímetro.
- Superficie.
- Suavidad (variación local en longitudes de radio).
- Compacidad ($\frac{perímetro^2}{área - 1{,}0}$).
- Concavidad (gravedad de las partes cóncavas del contorno).
- Puntos cóncavos (número de partes cóncavas del contorno).
- Simetría.
- Dimensión fractal ("aproximación al litoral" -1).

La media, el error estándar y la media de los tres mayores valores de estas características se han calculado para cada imagen, dando como resultado 30 características. Por ejemplo, el atributo 3 es el radio medio, el atributo 13 es el error estándar del radio y el atributo 23 la media de los tres mayores valores de radio.

Visualicemos la distribución de los datos respecto de los 10 primeros atributos de dos en dos; escogemos los 10 primeros ya que son los valores que representan la media de cada uno de los atributos.

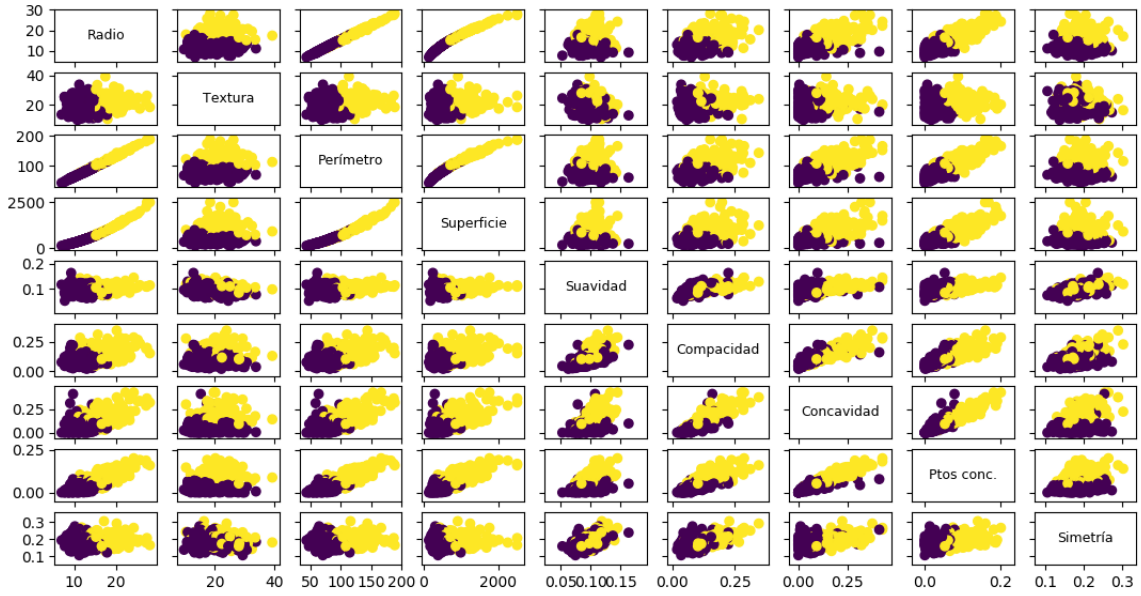

Figura 1: Gráficas de los diez primeros atributos de nuestra base de datos.

Viendo la gráfica podemos apreciar que existe una relación fuerte de dependencia entre algunas variables, como por ejemplo entre el radio y la superficie o el perímetro, o la concavidad y la compacidad.

En total, dispondremos de 357 instancias benignas y 212 instancias malignas (569 instancias). Como ya se habrá podido observar nos encontramos ante un problema de clasificación binaria.

2. Lectura y preprocesado de los datos.

Una vez entendida la naturaleza de nuestros datos, pasemos al preprocesado de éstos. Para comenzar, leeremos nuestros datos con las siguientes funciones facilitadas por `numpy`:

```
y = np.genfromtxt('datos/wdbc.data', delimiter=',', usecols=1, dtype=str)
X = np.genfromtxt('datos/wdbc.data', delimiter=',', usecols=range(2, 32))
```

De esta manera crearemos dos vectores, donde `y` contendrá las etiquetas del dataset (M para maligno, B para benigno) y `X` los atributos de éste. Notar que no guardamos los valores de la columna 0, pues ésta solo contiende el ID de la instancia, irrelevante para nuestro ajuste.

Llegados a este punto, dejemos claro que usaremos en un primer momento **regresión logística** para ajustar nuestros datos; y para ajustar dicho modelo, necesitaremos un conjunto de entrenamiento y otro de test.

Dado que solo disponemos de un conjunto de datos sin que estén separado en train y test, nos encargaremos de que una cuarta parte de éstos sean el conjunto test y el resto el conjunto de entrenamiento para el ajuste del modelo. Usaremos el siguiente código:

```
X_train, X_test, y_train, y_test =
    train_test_split(X, y, stratify=y, test_size=0.25, random_state = 0)
```

Los atributos de la función (perteneciente a `sklearn`) serán, por orden, los arrays de datos y las etiquetas, `stratify=y` lo cuál hará que la separación en train y test mantenga la proporción de etiquetas del conjunto original, `text_size=0.25`, es decir, un cuarto del total, y `random_state=0` que fijará una semilla para la separación aleatoria de los datos.

Si mostramos por pantalla una instancia de nuestro conjunto de datos obtendremos lo siguiente:

```
X[0] = [1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
       1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
       6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
       1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
       4.601e-01 1.189e-01]
```

Como podemos comprobar, el valor que toman los atributos es muy dispar, están muy esparcidos, teniendo valores entre 10^{-3} y 10^3, por lo que será necesario normalizar el valor de nuestros atributos.

Para ello usaremos la función `scale(X)` de `sklearn`, la cual estandariza un conjunto de datos centrándose en la media y en la escala de componentes con varianza unitaria, $\sigma^2 = 1$.

```
X_train = preprocessing.scale(X_train)
X_test = preprocessing.scale(X_test)
```

Si se hubiese aplicado la normalización sobre el conjunto de datos completo, sin haberlo separado en train y test, los datos de train tendrían, por decirlo de algún modo, cierta información sobre los datos del test. Debido a esto la hemos realizado tras el separado en train y test.

De esta manera, nuestros datos pasarán a ser de la siguiente forma:

```
X[0] = [1.09706398 -2.07333501  1.26993369  0.9843749   1.56846633  3.28351467
       2.65287398  2.53247522  2.21751501  2.25574689  2.48973393 -0.56526506
       2.83303087  2.48757756 -0.21400165  1.31686157  0.72402616  0.66081994
       1.14875667  0.90708308  1.88668963 -1.35929347  2.30360062  2.00123749
       1.30768627  2.61666502  2.10952635  2.29607613  2.75062224  1.93701461]
```

3. Modelo lineal - Regresión Logística.

Para empezar, trataremos de ajustar nuestra base de datos mediante un modelo lineal, en concreto mediante Regresión Logística.

Generados ya nuestro conjuntos train y test, es el momento de tratar la validación y regularización de los datos.

Con el conjunto de entrenamiento, usaremos validación y regularización para evaluar nuestro clasificador y calcular `c` (inversa de la "*regularization strenght*" (fuerza de regularización)), evitando así el sobreajuste de nuestro clasificador; en concreto, usaremos Validación Cruzada *K-fold*.

Para comprobar la necesidad de regularización, crearemos varios modelos, cada uno de ellos con un "*regularization strenght*" $\lambda = \frac{1}{c^i}$, donde i tomará valores enteros del intervalo $[-7, 7]$.

Para cada conjunto test obtenido mediante la validación cruzada, calcularemos el acierto respecto del ajuste realizado con la variable `c`, calculando una media de aciertos por cada `c`. Finalmente, escogeremos el parámetro `c` que mejores resultados nos haya dado.

La implementación es más sencilla de lo que aparenta; empezaremos creando una variable que contendrá los parámetros que queremos ajustar y los valores que puedan tomar; tras ello, usaremos la función `GridSearchCV(...)`

que creará los modelos para la validación cruzada. Finalmente solo nos quedará ajustar el modelo a partir de los datos, de manera que se asignará automáticamente el valor de `c` con menor error.

```
param_dist = {'C': np.float_power(10, range(-7,7))}
lr = GridSearchCV(LogisticRegression(), cv = 10, param_grid=param_dist)

# Ajustamos el modelo a partir de los datos
lr.fit(X_train, y_train)
```

Si ajustamos el modelo respecto de nuestros datos (llamando a la función `fit`), podremos comprobar el valor de `c`, el cuál será 1.

Realizados ya los cálculos sobre los datos y los modelos a usar, tenemos que el único parámetro usado ha sido `c`, el cual ha ido cambiando creando un nuevo modelo de regresión logística. Respecto a los hyperparámetros (parámetro cuyo valor se establece antes de que comience el proceso de aprendizaje), hemos fijado el valor $k = 10$, el cual establece el número de subconjuntos a generar mediante la validación cruzada.

Ya tenemos todo lo necesario para realizar nuestros cálculos y determinar el porcentaje de acierto para nuestro modelo. Como modelo final, seleccionaremos la regresión logística con el parámetro `c` que mejor porcentaje medio nos haya dado en la regularización, es decir, `c` $= 1$. Dicho modelo ya ha sido ajustado anteriormente mediante `lr.fit(X_train, y_train)`.

A continuación, calculemos el porcentaje de acierto para los datos de entrenamiento y los datos de test. Bastará con el siguiente código:

```
# Calculamos el score con dicho ajuste para test
predictions_train = lr.predict(X_train)
score_train = lr.score(X_train, y_train)

# Calculamos el score con dicho ajuste para test
predictions_test = lr.predict(X_test)
score_test = lr.score(X_test, y_test)

print('Mejor C: ', lr.best_params_['C'])
print('Valor de acierto con el mejor c sobre el conjunto train: ', score_train)
print('Valor de acierto con el mejor c sobre el conjunto test: ', score_test)
input("Pulsa enter para continuar.")
```

Tras ejecutar ésto, obtendremos la siguiente salida:

```
Mejor valor de C:  1.0
Valor de acierto con el mejor c sobre el conjunto train:  0.9929577464788732
Valor de acierto con el mejor c sobre el conjunto test:  0.986013986013986
Pulsa enter para continuar.
```

Por tanto, el modelo elegido realiza un buen ajuste de los datos, pues estamos obteniendo un porcentaje de precisión del 98.6 % sobre los datos de test, un valor de acierto bastante alto.

La matriz de confusión obtenida es:

$$P = \begin{bmatrix} 90 & 0 \\ 2 & 51 \end{bmatrix}$$

Como podemos ver, el modelo solo ha fallado dos veces, cometiendo dos falsos negativos; así, podemos reiterar que el ajuste calculado es bueno.

Veamos a continuación la curva ROC asociada al modelo. Para calcularla y mostrarla por pantalla, usaremos el siguiente código:

```
y_pred_rf = lr.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_rf)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='darkorange', label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
```

```
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

Finalmente, la curva generada será:

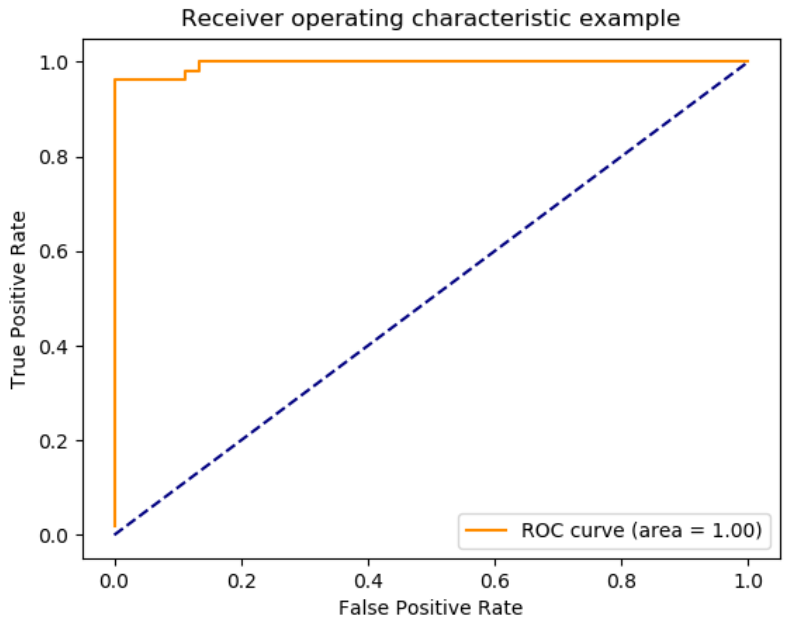

Figura 2: Curva ROC del ajuste mediante Regresión Logística.

Como podemos ver, el área formada bajo la curva vale 1, interpretando ésto como que el ajuste del modelo es excelente.

Por tanto, la regresión logística será un buen clasificador sobre nuestro conjunto de datos.

4. Modelos no-lineales - Random Forest.

Para Random Forest se han tomado los datos sin realizar ningún tipo de escalado o modificación sobre estos.

Lo primero que vamos a hacer es utilizar Random Forest para la obtención de las características mas importantes. Para ello, definiremos un clasificador y entrenaremos con el conjunto de datos train.

```
clf = RandomForestClassifier(random_state=10)
clf.fit(X_train, y_train)
```

Ahora obtenemos la importancia de cada característica y mostramos una gráfica con dichas características ordenadas por importancia.

```
importances = clf.feature_importances_
std = np.std([tree.feature_importances_ for tree in clf.estimators_], axis=0)
indices = np.argsort(importances)[::-1]

plt.figure()
plt.title("Feature importances")
```

```
plt.bar(range(X.shape[1]), importances[indices],
color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

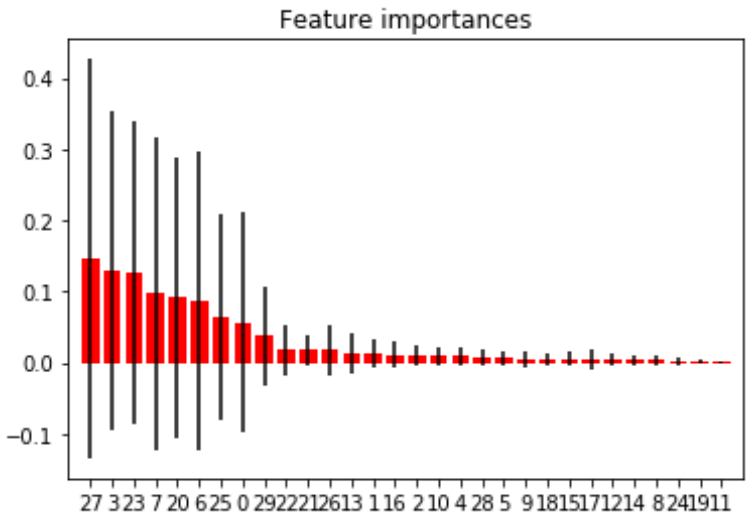

Figura 3: Importancia de las características

Una vez obtenida la importancia de cada característica, nos vamos a quedar con aquellas que tengan un valor más alto.

```
features_to_use = indices[0:14]

X_train = X_train[:,features_to_use]
X_test = X_test[:,features_to_use]
```

Una vez hecho esto, pasamos a la obtención del modelo de Random Forest. Para ello tomaremos un modelo en el que vamos a fijar el número de características a $\sqrt{n}$, donde n es el número de características. Esto se consigue con el parámetro `max_features = 'sqrt'`. También fijaremos el valor de la semilla, `random_state=10`.

```
fit_rf = RandomForestClassifier(max_features = 'sqrt', random_state=10)
```

El parámetro que vamos a ajustar en Random Forest es `n_estimators`, que se corresponde con el número de arboles en el *forest* generado. Para ello vamos a hacer uso de `GridSearchCV(...)` igual que hicimos en con el modelo de regresión logística.

```
estimators = range(10,200,10)
param_dist = {'n_estimators': estimators}
clf= GridSearchCV(fit_rf, cv = 10, param_grid=param_dist, n_jobs = 3)
```

Como podemos ver, el número de árboles lo hemos modificado desde 10 hasta 200, realizando saltos de 10. En el `GridSearchCV(...)` el parámetro `cv=10` indica que se va a hacer una validación cruzada con $K = 10$; y el parámetro `n_jobs=3`, que se ejecutarán en paralelo 3 tareas.

Tras esto ajustamos el modelo con los datos de entrenamiento.

```
clf.fit(X_train, y_train)
```

Después del ajuste, obtenemos los scores obtenidos de media en las validaciones cruzadas, para cada valor del parámetro `n_estimators`, para representar la variación del error de test al aumentar el número de árboles.

```
scores = clf.cv_results_['mean_test_score']
plt.plot(estimators, 1-scores)
plt.xlabel('num tree')
plt.ylabel('test error')
plt.show()
```

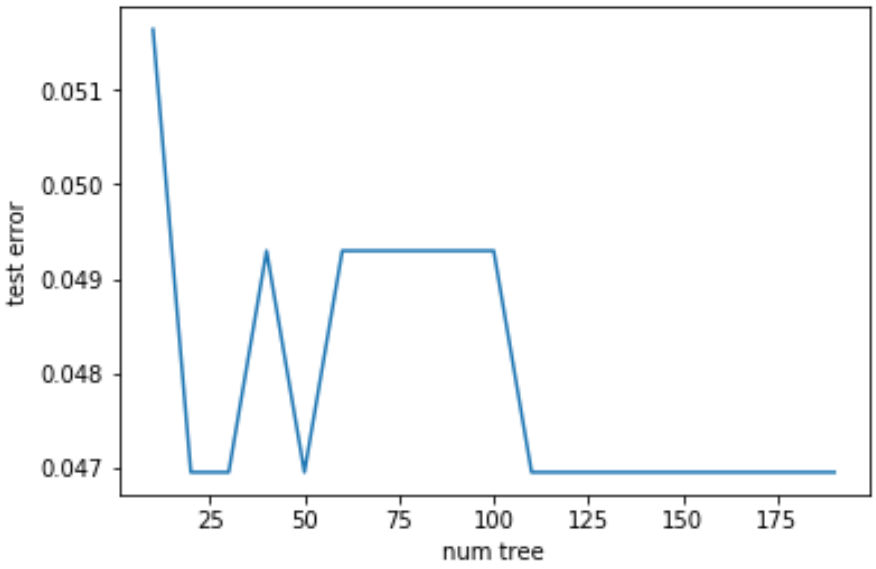

Figura 4: Variación del score respecto `n_estimators`.

Vemos ahora cual ha sido el mejor valor del parámetro:

```
best_param = clf.best_params_['n_estimators']
print ("Mejor valor para n_estimators: ", best_param)
```

La salida obtenida es: `Mejor valor para n_estimators: 20`.

Finalmente, comprobemos la precisión obtenida en el train y en el test.

```
predictions_train = clf.predict(X_train)
predictions = clf.predict(X_test)

print ("Train Accuracy :: ", accuracy_score(y_train, predictions_train))
print ("Test Accuracy  :: ", accuracy_score(y_test, predictions))
print (" Confusion matrix \n", confusion_matrix(y_test, predictions))
```

La salida obtenida es:

```
Train Accuracy :: 0.9976525821596244
Test Accuracy :: 0.986013986013986
Confusion matrix
[[90  0]
[2  51]]
```

Los valores de precisión obtenidos han sido muy altos, de hecho casi perfectos. En la matriz de confusión podemos apreciar que tan solo ha cometido dos errores de falsos negativos, acertando todas las demás predicciones. Comentar también que el error ha sido en un falso negativo, es decir diagnosticar a una persona con tumpo maligno cuando en realidad era benigno, este error es menos grave que el otro que se podría haber cometido, el de diagnosticar como benigno cuando en realidad era maligno.

Por último representemos la curva ROC e interpretemos los resultados. La curva ROC obtenida es la siguiente. Como podemos ver el área bajo la curva es de 0.99, este valor indica que el test ha sido excelente, por lo que podemos considerar que el modelo obtenido es bastante bueno.

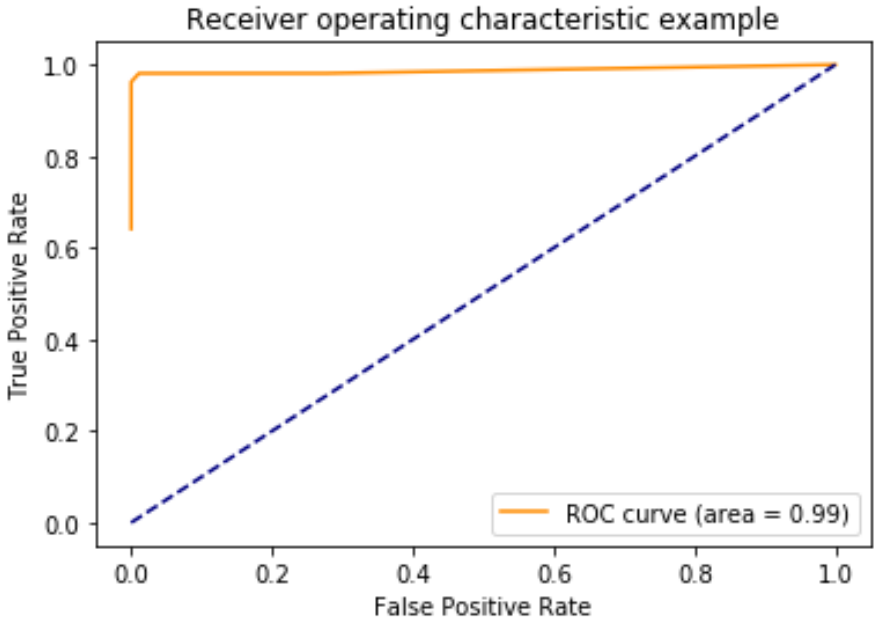

Figura 5: Curva ROC del ajuste mediante Random Forest.

5. Modelos no-lineales - Support Vector Machine.

Hablemos a continuación sobre el ajuste mediante SVM; en concreto, usaremos un núcleo polinomial.

SVM se encargará de separar los datos mediante un hiperplano de manera que la distancia de los puntos más cercanos a éste (vectores de soporte) sea óptima, es decir, la más grande posible.

Para el ajuste mediante este clasificador tomaremos los datos con el mismo escalado que hemos tratado en el apartado *Lectura y preprocesado de los datos*. Además, usaremos validación y regularización como en los anteriores apartados.

Supuesto que ya hemos leído, preprocesado y realizados todos los cálculos sobre nuestros datos previos al ajuste, comenzamos con la Validación Cruzada *K-fold*. Esta vez, en vez de aplicar regularización sobre un único parámetro, la aplicaremos sobre dos; en concreto sobre el parámetro `C`, similar al ajustado en Regresión logística, y el parámetro `degree`, el cual denotará el grado de la función polinómica del kernel de SVM. Utilizaremos 10 *folds*.

Para ello, usaremos el siguiente código:

```
# Validacion y regularizacion
c_range = np.float_power(10, range(-7,7))
degree_range = list(range(1,5))
param = dict(degree=degree_range, C=c_range)
svmachine=svm.SVC(kernel='poly', probability=True)
clf = GridSearchCV(svmachine, cv = 10, param_grid=param)
```

Podemos comprobar que el valor de `C` oscilará entre 10^{-7} y 10^{7}, mientras que el polinimio será de grado entre 1 y 4. Además, establecemos el kernel polinomial en la llamada a SVC con `kernel='poly'`, y establecemos `probability=True` para poder representar más tarde la curva ROC.

Tras esto, ajustamos el modelo sobre nuestros datos de entrenamiento y nos dispondremos a representar la variación de la precisión de nuestro modelo respecto del parámetro `C` y el grado del polinomio. El siguiente código será el encargado.

```
# Ajustamos el modelo a partir de los datos
clf.fit(X_train, y_train)

# Dibujamos las gráficas en función de C y degree
params = clf.cv_results_['params']
scores = clf.cv_results_['mean_test_score']
```

```
plt.plot(c_range, scores[0::4], 'r-', label='grado 1')
plt.plot(c_range, scores[1::4], 'b-', label='grado 2')
plt.plot(c_range, scores[2::4], 'g-', label='grado 3')
plt.plot(c_range, scores[3::4], 'y-', label='grado 4')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('score')
plt.legend()
plt.show()
```

El resultado obtenido tras la ejecución de estas líneas será el que mostramos a continuación:

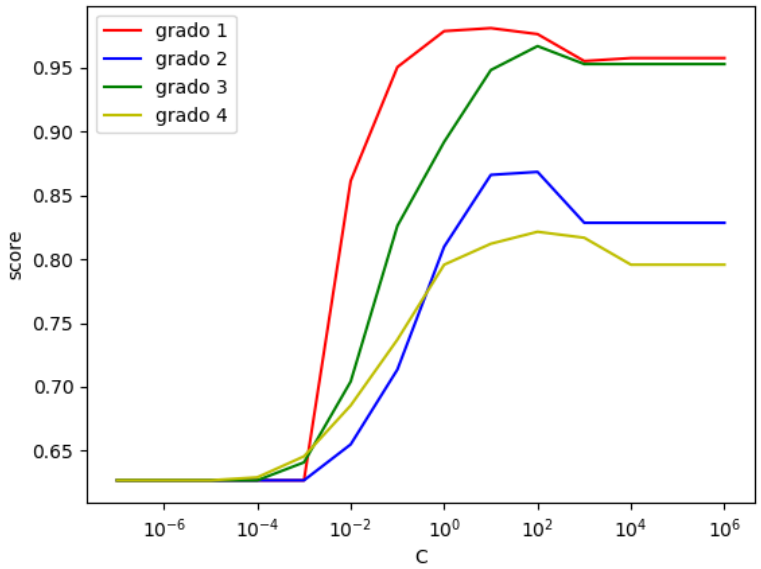

Figura 6: Variación de la precisión del modelo respecto del valor de `C`.

Es fácil observar que el grado óptimo para el núcleo polinómico será 1 y que a mayor grado mayor sobreajuste; por otra parte, vemos que el valor máximo de precisión se alcanzará aproximadamente en `C=10`. Recordar que el modelo lineal de Regresión logística ajustaba muy bien nuestros datos, por lo que es entendible que el grado óptimo para el núcleo de SVM sea 1.

Una vez realizada la validación y regularización, pasemos al ajuste final del modelo. Imprimiremos por pantalla los valores óptimos para `C` y `degree`, así como el número de vectores de soporte para cada clase de nuestro conjunto de datos. Por último, calcularemos la precisión con los datos de entrenamiento y los datos test.

```
# Calculamos el score con dicho ajuste para test
predictions_train = clf.predict(X_train)
score_train = clf.score(X_train, y_train)

# Calculamos el score con dicho ajuste para test
predictions_test = clf.predict(X_test)
score_test = clf.score(X_test, y_test)

print('\nMejor valor de C y mejor grado: ', clf.best_params_)
print('Número de vectores de soporte para cada clase: ', clf.best_estimator_.n_support_)
print('Valor de acierto con el mejor c sobre el conjunto train: ', score_train)
print('Valor de acierto con el mejor c sobre el conjunto test: ', score_test)
input("Pulsa enter para continuar.")
```

Dicho código devolverá la siguiente información:

```
Mejor valor de C y mejor grado:  {'C': 10.0, 'degree': 1}
```

```
Número de vectores de soporte para cada clase:  [19 21]
Valor de acierto con el mejor c sobre el conjunto train:  0.9906103286384976
Valor de acierto con el mejor c sobre el conjunto test:  0.986013986013986
Pulsa enter para continuar.
```

Por tanto, hemos obtenido un gran ajuste, con prácticamente la misma precisión que los anteriores modelos, un 98.6 %.

La matriz de confusión asociada a este clasificador es exactamente la misma que en los dos modelos anteriores, únicamente con dos falsos negativos. Respecto a la curva ROC, tenemos la siguiente gráfica:

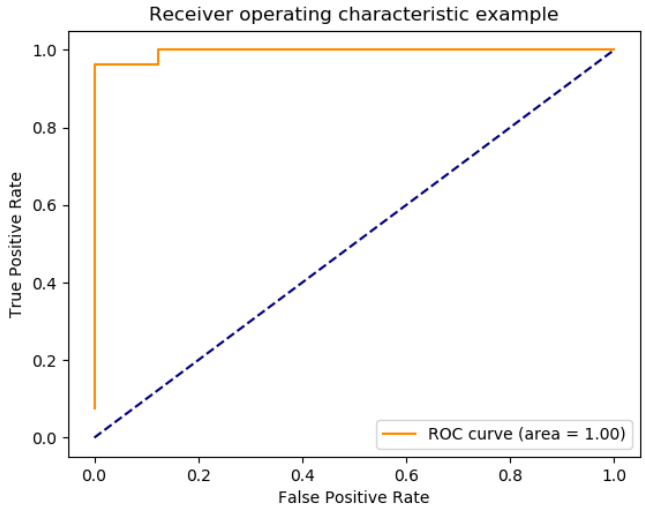

Figura 7: Curva ROC del ajuste mediante SVM.

6. Modelos no-lineales - Redes Neuronales

Procedemos igual que siempre, leyendo los datos y modificando las etiquetas para pasarlas a valores numéricos. A continuación separamos los datos en train y test.

```
# Leemos los datos
y = np.genfromtxt('datos/wdbc.data', delimiter=',', usecols=1, dtype=str)
X = np.genfromtxt('datos/wdbc.data', delimiter=',', usecols=range(2, 32))

le = preprocessing.LabelEncoder()
y = le.fit(y).transform(y)

# Separamos los datos en train y test
X_train, X_test, y_train, y_test = train_test_split(X, y,
 stratify=y, test_size=0.25, random_state = 0)
```

Perceptron multicapa es sensible al escalado de funciones. Hemos decidido estandarizar los datos para que tengan la media 0 y la varianza 1. Esto se ha hecho mediante StandardScaler.

```
X_train = StandardScaler().fit_transform(X_train)
X_test = StandardScaler().fit_transform(X_test)
```

Lo primero que vamos a hacer es elegir el número de capas ocultas de la red neuronal y el número de unidades por capa. El número de capas va a variar entre 1 y 3, y el número unidades por capa entre 0 y 50. Como el número de posibles modelos entre estos rangos es muy alto, vamos a considerar que todas las capas tienen el mismo número de capas, por ejemplo una red con 3 capas ocultas y 10 unidades en cada capa. Los modelos de perceptrones multicapa los vamos a definir con `MLPClassifier` y el parámetro que vamos a modificar `hidden_layer_sizes`.

El parámetro `hidden_layer_sizes` se fija pasandole una tupla, por ejemplo (5,8,3) significaría que hay tres capas ocultas, donde a primera capa tendría 5 unidades, la segunda 8 y la tercera 3.

A continuación se crea un vector con las tuplas que se van a usar para el parámetro `hidden_layer_sizes`.

```
hls = []
for j in range(0,3):
   for i in range(1,50,5):
      v = []
      for k in range(0,j+1):
         v.append(i)
      hls.append(v)

print("hidden_layer_sizes:\n", hls)
```

Las tuplas anteriormente generadas se corresponden con:

```
[[1], [6], [11], [16], [21], [26], [31], [36], [41], [46], [1, 1],
[6, 6], [11, 11], [16, 16], [21, 21], [26, 26], [31, 31], [36, 36],
[41, 41], [46, 46], [1, 1, 1], [6, 6, 6], [11, 11, 11], [16, 16, 16],
[21, 21, 21], [26, 26, 26], [31, 31, 31], [36, 36, 36], [41, 41, 41],
[46, 46, 46]]
```

Para ajustar el hiperparámetro `hidden_layer_sizes`, vamos a usar validación cruzada manteniendo la proporción de las clases. El valor de k va a ser igual a 5.

```
# KFold
k = 5
kf = StratifiedKFold(n_splits=k)
kf.get_n_splits(X_train,y_train)

mejor = 0
mejor_media = 0

for i in range(0,len(hls)):
   suma = 0
   for train_index, test_index in kf.split(X_train,y_train):
      X_train_, X_test_ = X_train[train_index], X_train[test_index]
      y_train_, y_test_ = y_train[train_index], y_train[test_index]

      mlp = MLPClassifier(max_iter=500, hidden_layer_sizes = hls[i], random_state = 10)
      mlp.fit(X_train_, y_train_)
      suma += mlp.score(X_test_, y_test_)

   media = 1.0*suma/k

   if media > mejor_media:
      mejor_media = media
      mejor = i

print(hls[mejor])
```

En este proceso se obtuvo el siguiente warning

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (500) reached and the
optimization hasn't converged yet. % self.max_iter, ConvergenceWarning)
```

Debido a que con un número máximo de 500 iteraciones no llega a converger, se arreglaría facilmente aumentando las iteraciones máximas hasta que no salte el warning, pero no se ha hecho por temas de tiempo de ejecución. Anteriormente hicimos pruebas con 200 iteraciones como máximo que era el valor que venía por defecto y se

obtuvieron muchos warning, motivo por el cual se aumento el número máximo de iteraciones 500, reduciendo el número de warning a solo 1.

Como salida se ha obtenido `Mejor hidden_layer_sizes: [31]`, es decir el que mejores resultados a dado en la validación cruzada, ha sido el modelo de solo una capa oculta con 31 unidades.

Tras la obtención de la mejor arquitectura pasamos a ajustar el parámetro de regularización alpha. Lo hacemos igual que otras veces con GridSearchCV, en donde hemos usado Kfold con k=10. Y un valor de max_iter = 500, para evitar tener warning en la convergencia.

```
model = MLPClassifier(max_iter=1000, hidden_layer_sizes = hls[mejor], random_state = 10)

param = {'alpha': 10.0 ** -np.arange(1, 7)}
mlp = GridSearchCV(model, cv = 10, param_grid=param, n_jobs = 3)
mlp.fit (X_train, y_train)

print ("\nMejor valor de alpha: ", mlp.best_params_['alpha'])
```

Obtenemos: `Mejor valor de alpha: 0.1`

Ahora pasamos a comprobar la precisión de nuestro modelo en los datos del train y test. Y mostramos la matriz de confusión obtenida.

```
predictions_train = mlp.predict(X_train)
predictions = mlp.predict(X_test)

print ("Train Accuracy :: ", accuracy_score(y_train, predictions_train))
print ("Test Accuracy  :: ", accuracy_score(y_test, predictions))
print (" Confusion matrix \n", confusion_matrix(y_test, predictions))
```

Donde obtenemos como salida:

```
Train Accuracy ::  0.9929577464788732
Test Accuracy  ::  0.986013986013986
 Confusion matrix
 [[90  0]
  [ 2 51]]
```

Los valores de precisión obtenidos son muy buenos tanto en el train como en el test. Si nos fijamos hemos vuelto a obtener la misma precisión en el test que en los modelos anteriores y la misma matriz de confusión.

Ahora veamos la curva de ROC obtenida.

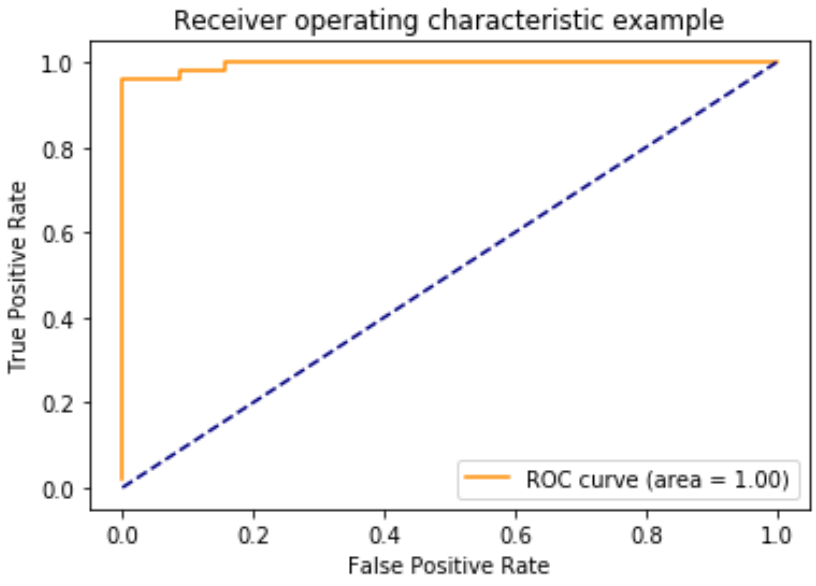

Figura 8: Curva ROC

De nuevo obtenemos una curva Roc muy buena lo cual indica un test muy bueno, de donde podemos deducir que el modelo tiene alta probabilidad de ser bueno.

7. Conclusión

Después de probar todos los modelos hemos llegado a los mismos resultados para el test en cada modelo, y con curvas de Roc casi iguales, habiendo ajustado lo mejor posible cada modelo, escalando los datos en ocasiones, estandarizandolos en el caso de las redes neuronales, eliminando características poco importantes obtenidas en random forest. También hemos evitado el sobreentrenamiento, mediante validación cruzada y probando distintos parámetros de regularización. Por lo que finalmente nos quedamos con el primer modelo, el modelo lineal, regresión logística, ya que nos ha dado unos resultados excepcionales y ha igualado a los demás modelos probados, siendo regresión lineal el modelo más sencillo de todos. Mencionar que en la descripción de la base de datos decía que los datos se habían conseguido ajustar perfectamente mediante un modelo lineal, cosa que ha quedado comprobada con los experimentos realizados.