UNIVERSIDAD
DE GRANADA

Aprendizaje Automático
grado en ingeniería informática

PROYECTO FINAL

Subtítulo práctica

Autores
Vladislav Nikolov Vasilev
José María Sánchez Guerrero

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2018-2019

Índice

1. Descripción del problema

El conjunto de datos con el que vamos a trabajar es el *Image Segmentation Data Set*, creado por el Vision Group de la Universidad de Massachusetts. Este conjunto de datos contiene una serie de características de casos extraídos al azar de una base de datos con 7 tipos de imágenes tomadas al aire libre. Éstas imágenes fueron segmentadas manualmente para crear una clasificación para cada píxel. Cada instancia es una región de 3×3.

Originalmente, nuestro conjunto de datos estaba ya dividido en training y test. Sin embargo, disponemos solo de 210 datos de entrenamiento y 2100 de test. Como la cantidad de datos que tenemos para entrenamiento es muy inferior a la de test, y no existe un motivo justificado por el que las particiones se hayan hecho de esta forma, hemos decidido juntar todos los datos y crear nuestras propias particiones de entrenamiento y test. Esta división se comentará con más detalles en secciones posteriores, cuando hablemos de las transformaciones y preprocesado que hemos realizado sobre los datos de los que disponemos.

Según la información proporcionada por la descripción del conjunto de datos, la cuál puede ser encontrada en el repositorio UCI [**?**], existen 7 clases distintas, y hay 30 muestras de cada clase en el conjunto de entrenamiento y 300 en el conjunto de test. Con lo cuál, tenemos que en total hay 330 muestras de cada clase si miramos los dos conjuntos de datos de forma conjunta. En total disponemos de 2310 muestras, cada una de las cuáles tiene 19 atributos que toman valores reales. En la representación original de los datos, tenemos que la primera columna se corresponde con la clase de la muestra, y las 19 columnas restantes se corresponden con los atributos.

De esta pequeña descripción podemos concluir que cada clase está idénticamente representada en los datos de los que disponemos, y que no existe una clase que esté representada en mayor o menor medida que el resto de ellas.

Con todo esto dicho, vamos a proceder a analizar cada una de las 19 características mencionadas anteriormente, para ver que representa cada una de ellas:

1. *Region-centroid-col*: la columna del píxel central de la región.
2. *Region-centroid-row*: la fila del píxel central de la región.
3. *Region-pixel-count*: el número de píxeles en una región. Su valor siempre es 9.
4. *Short-line-density-5*: resultados de un algoritmo de extracción de rectas que cuenta cuántas líneas de longitud 5 (con cualquier orientación) con bajo contraste, menor o igual a 5, cruzan la región.

5. *Short-line-density-2*: igual que *Short-line-density-5* pero cuenta con líneas de alto contraste, mayor que 5.
6. *Vegde-mean*: mide el contraste de píxeles horizontalmente adyacentes en la región. Hay 6 valores, pero se da dan la media y la desviación típica. Este atributo se utiliza como un detector de borde vertical.
7. *Vegde-sd:* Desviación típica del contraste de píxeles horizontalmente adyacentes en la región (ver **??**).
8. *Hedge-mean*: mide el contraste de los píxeles verticalmente adyacentes. Utilizado para la detección de líneas horizontales. Este atributo es el valor medio.
9. *Hedge-sd*: Desviación típica del contraste de los píxeles verticalmente adyacentes (ver **??**).
10. *Intensity-mean*: el promedio sobre la región de (R + G + B) / 3.
11. *Rawred-mean*: Promedio sobre la región del valor R.
12. *Rawblue-mean*: Promedio sobre la región del valor B.
13. *Rawgreen-mean*: Promedio sobre la región del valor G.
14. *Exred-mean*: Mide el exceso de rojo: (2R - (G + B)).
15. *Exblue-mean*: Mide el exceso de azul: (2B - (G + R)).
16. *Exgreen-mean*: Mide el exceso de verde: (2G - (R + B)).
17. *Value-mean*: Transformación 3D no lineal de RGB.
18. *Saturatoin-mean*: (ver **??**).
19. *Hue-mean*: (ver **??**).

Como estamos en un problema de clasificación, cada entrada produce una salida que es una etiqueta. La etiqueta puede hacer referencia a una de las 7 clases que tiene el problema, las cuáles son: *brickface*, *sky*, *foliage*, *cement*, *window*, *path* y *grass*. Para facilitar la representación de las clases, vamos a transformar posteriormente cada valor de las etiquetas a un valor numérico. Esto se verá con más detalle en la siguiente sección, en la que hablaremos del preprocesado realizado sobre los datos. continuación.

2. Preprocesado de los datos de entrada

En esta sección vamos a comentar el preprocesado que hemos hecho a los datos de entrada. Es decir, vamos a ver cómo y por qué juntamos los dos conjuntos de datos y luego los separamos (tal y como dijimos en la sección anterior) y vamos a ver que modificaciones hacemos sobre éstos para que nos sea más fácil trabajar posteriormente con ellos y para adaptarlos mejor al problema que queremos resolver.

Vamos a comenzar leyendo los datos y almacenándolos en estructuras de datos que nos permitan el fácil acceso a éstos. Posteriormente, tal y como dijimos anteriormente, vamos a juntar los dos conjuntos en uno solo. Esto se puede ver a continuación:

```
# Leer los datos de training y test
df1 = read_data_values('datos/segmentation.data')
df2 = read_data_values('datos/segmentation.test')

# Juntar los dos conjuntos en uno solo
df = pd.concat([df1,df2])
```

Nuestro objetivo al hacer esto es disponer de todos los datos de forma conjunta para luego poder crear nuestras propias particiones de entrenamiento y de test, debido a que, como hemos dicho anteriormente, la cantidad de datos de entrenamiento de la que disponemos es muy pequeña. Como no existe una justificación para no crear nuestras propias particiones, vamos a crearlas de tal forma que tengamos más datos de entrenamiento que de test, conservando además la forma en la que cada clase está representada. Con esto, obtendremos un mejor modelo, ya que tendremos más datos con los que entrenarlo, y por tanto, va a generalizar mejor.

Sin embargo, antes de realizar las particiones, vamos a transformar las clases originales a valores numéricos, para que el trabajar con ellas posteriormente sea más sencillo. Vamos a asignar a cada etiqueta un número. Como tenemos 7 clases, asignaremos un número en el rango $[0, 6]$ a cada una de ellas, de forma que no se repita ningún número para ninguna etiqueta. Esta asignación se puede ver a continuación:

```
# Valor numerico asignado a cada etiqueta
labels_to_values = { 'BRICKFACE' : 0, 'SKY' : 1, 'FOLIAGE' : 2,
                     'CEMENT': 3, 'WINDOW' : 4, 'PATH' : 5,
                     'GRASS' : 6 }
```

Ahora ya solo nos quedaría aplicar el *mapping*. Como todavía no hemos separado las etiquetas de los datos y sabiendo que, gracias a la información proporcionada por el repositorio, la etiqueta se encuentra en la primera columna, la sustitución se puede hacer de forma sencilla de la siguiente manera:

```
# Sustituir etiquetas de salida por valores numericos discretos
df[0] = df[0].map(labels_to_values)
```

Con esto ya hecho, ya podemos dividir los datos en los valores de entrada $\mathcal{X}$ y las etiquetas y y posteriormente dividir éstos en los conjuntos de entrenamiento y de test. Queremos que el 80 % de los datos de los que disponemos esté en el conjunto de entrenamiento y que el 20 % restante esté en el de test. Además, tal y como se mencionó anteriormente, queremos que las clases estén igualmente representadas en los dos conjuntos en función de la cantidad de muestras que hay en total de cada clase. También es importante mezclar los datos, con tal de que la elección de qué muestra va a cada conjunto no sea decidida por el orden de las muestras, si no que se haga de forma aleatoria.

Se ha escogido esta repartición de los datos ya que nos permite tener un número suficientemente grande de muestras para entrenar nuestro modelo y una cantidad aceptable de datos para ver como se desempeña con datos nunca vistos antes. La desventaja de hacer *hold-out* (que es lo que estamos haciendo aquí) es que perdemos datos para entrenar nuestro modelo, datos con los que, muy posiblemente, pudiésemos obtener unos mejores resultados. Sin embargo, siguiendo este enfoque, tendremos al menos una forma de ver cómo de bien funciona nuestro modelo con nuevos datos con los que nunca antes había interaccionado.

Vamos a ver ahora como se realizaría esta partición de los datos:

```
# Obtener valores X, Y
X, y = divide_data_labels(df)

# Dividir los datos en training y test
# Conservar proporcionalidad de clase mezclando los datos
print('Splitting data in training and test sets...')
X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=1, shuffle=True, stratify=y)
```

Y con todo esto visto, pasemos ahora a la parte de análisis de los datos, la cuál nos va a permitir obtener más información acerca de los datos de los que disponemos.

3. Análisis de los datos

Para comenzar con el análisis de los datos que tenemos, lo primero tenemos que hacer es, obviamente leer los datos. Para ello, vamos a comenzar cargándolos en estructuras que nos permita almacenar tanto los datos de entrenamiento como los de test para posteriormente juntarlos en una única:

Con los datos ya cargados, nos interesa Posteriormente, tal y como dijimos en la sección anterior, vamos a juntar los datos en un solo conjunto para dividirlos y crear nuestros propios conjuntos de entrenamiento y de test. para poder posteriormente juntar los datos, tal y como dijimos que haríamos en la sección anterior por disponer de demasiados poco, y posteriormente los vamos a juntar, tal y como dijimos en la sección anterior.

4. Elección de la mejor técnica

Una vez analizados todos los modelos, utilizando **cross-validation** y diversos hiperparámetros para cada uno de ellos, vamos a comentar cuáles han sido los que han obtenido los mejores resultados y porqué. Para ello nos vamos a ayudar de las métricas obtenidas anteriormente, es decir, utilizaremos tanto la tabla que nos muestra la **media de aciertos** y **desviaciones típicas**, como las **curvas de aprendizaje** de cada uno de los modelos (poner referencia a las tablas y gráficas).

Observando los resultados obtenidos, tenemos que el Random Forest con un $n\_estimators = 50$ es el que mayor porcentaje de acierto ha tenido, con un 97.8470 %; mientras que el que menor desviación típica ha conseguido es el MLP-Classifier con un $hidden\_layer\_sizes = (100 - 100)$, con un valor de 0.005839. No obstante, podemos ver que el porcentaje de acierto de la mayoría de clasificadores es muy alto, entre el 96 % y el 97 %, y los valores de las desviaciones típicas también son bastante bastante bajos, entre el 0.005 y el 0.02; así que tendremos que tener en cuenta más cosas a parte de estas.

Vamos a analizar ahora las gráficas de los modelos para ver si encontramos algunos detalles más significativos en ellas. Si empezamos por las de Regresión Logística, vemos que las curvas son bastante suaves (sobretodo para los parámetros de $C = 1.0$ y $C = 5.0$) y a medida que aumenta el número de evaluaciones, la clasificación va mejorando y el sobreajuste se reduce. En cuanto al SVM, podemos ver que las gráficas no son malas, es decir, son bastante suaves y apenas tienen sobreajuste, pero tampoco son las mejores. En el Random Forest podemos ver que las curvas van aumentando suavemente a medida que el número de evaluaciones es más grande, no obstante, (¿¿¿¿¿¿el valor del **Training score** ronda siempre el 1, lo que puede causar sobreajuste. Aún así, no podemos decir que sean malas?????). Finalmente, para las gráficas del MLPClassifier .....

Gracias a este análisis, podemos decir que el SVM no va a ser uno de nuestros mejores modelos, porque nos ha ofrecido los resultados más pobres en cuanto a la precisión y la desviación típica, y sus gráficas tampoco eran las mejores. Aun así, este no es un mal clasificador (hemos podido comprobar que su porcentaje de acierto suele ser mayor de 90), pero lo tenemos que descartar porque tenemos al resto que están un paso por delante.

Entre los tres modelos restantes nos vamos a quedar con la **Regresión Logística** y con el **Random Forest** por lo siguiente. Los tres modelos han obtenido unos resultados muy buenos, incluso me atrevería a decir que el MLPClassifier tiene las mejores curvas de aprendizaje y sólo superado en porcentaje de aciertos por algunos Random Forest, sin embargo, hemos descartado este clasificador porque es mucho más complejo que los otros dos. Viendo los resultados, no hay una

diferencia que sea lo suficientemente relevante como para elegir este modelo, que es mucho más difícil y costoso en cuanto a tiempo de cómputo; frente a la rapidez y simplicidad de la Regresión Logística, o frente al Random Forest que obtuvo los mejores resultados y que también es bastante más rápido.

5. Ajuste de las técnicas seleccionadas

5.1. Regresión Logística

Este modelo hemos visto que no ofrecía las mejores soluciones (aunque no se quedaba lejos de ellas), pero como es significativamente más rápido que las redes neuronales, vamos a intentar ajustas sus hiperparámetros al máximo para conseguir mejorar sus resultados. Esto lo haremos creando un grid con los distintos valores que le podremos asignar a nuestro algoritmo, y posteriormente ejecutándolo para cada uno de ellos. Este es código que crea el grid y el modelo de Regresión Logística:

```
# Crear grid de hiperparametros que se van a probar
param_grid_lr = [{'C': np.linspace(0.1, 1.0, 10),
                  'multi_class': ['multinomial'],
                  'solver': ['newton-cg'],
                  'random_state': [1]}]

# Crear modelo de regresion logistica
mlr = LogisticRegression()
```

Para el parámetro C generamos una muestra de tamaño 10 con valores que van del 0.1 hasta el 1.0. Éste será el único hiperparámetro que cambie, ya que *multi_class* sólo podrá adoptar el valor $'multinomial'$ y *solver* sólo tendrán el valor $'newton - cg'$. El primero de ellos es porque estamos en un problema multiclase, y el segundo de ellos porque ofrece una mejor minimización de la función cuadrática. El último hiperparámetro *random_state* es la semilla para los aleatorios, así que no es muy importante.

A continuación, utilizaremos el método ***GridSearchCV*** [?] para realizar una búsqueda exhaustiva entre todos los valores de los parámetros especificados y quedarnos con el mejor. Este método implementa funciones como *fit* o *score* que nos ayudarán a evaluarlo, y parámetros como *best_estimator_* que nos dirá cuál ha sido el mejor modelo de todos. Esta evaluación, al igual que las anteriores, también se realizará mediante *cross-validation* (implementada por la propia función). Veamos el código donde aplicamos *GridSearchCV*:

```
# Crear GridSearch con Cross Validation para determinar
# la mejor combinacion de parametros
grid_search = GridSearchCV(mlr, param_grid=param_grid_lr, cv=cv,
                           scoring='accuracy')
```

```
5
6 # Aplicar GridSearch para obtener la mejor combinacion de
      hiperparametros
7 grid_search.fit(X_train, y_train)
8
9 # Obtener indice de la mejor media de test
10 best_idx = np.argmax(grid_search.cv_results_['mean_test_score'])
```

El resultado de ejecutar todo esto ha sido el siguiente:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='multinomial',
          n_jobs=None, penalty='l2', random_state=1, solver='newton-cg',
          tol=0.0001, verbose=0, warm_start=False)

Mean training accuracy: 0.9618822212429274
Training accuracy Std. Dev: 0.0019126473836255846
Mean CV accuracy: 0.9556277056277056
CV accuracy Std. Dev: 0.012767395425462041
```

Figura 1: Resultados de *GridSearch* sobre la Regresión Logística.

Como podemos observar, el método ha selecciona el mismo valor para el hiperparámetro C que el que nosotros habíamos prefijado en las pruebas iniciales (1.0). Este valor es el que también trae la función por defecto, y acabamos de comprobar el porque. Valores más bajos hacen que la regularización sea muy fuerte, mientras que con valores más altos el sobreajuste aparecerá debido a que los errores tendrán una mayor penalización.

Si nos paramos a analizar tanto la precisión como la desviación típica hay poco que comentar. Como los hiperparametros utilizados han sido los mismos que en las pruebas iniciales, los valores que han obtenido han sido prácticamente iguales que los anteriores, con un decimal o dos de diferencia, pero prácticamente es insignificativo.

Concluimos que, pese a no haber mejorado porque ya habíamos elegido los valores más adecuados, el modelo sigue clasificando bastante bien. No obstante, no lo seleccionaríamos como el mejor modelo, ya que los clasificadores Random Forest lo hacen mejor con una complejidad similar (e incluso las Redes Neuronales, aunque habría que estudiarlo en términos de eficiencia y simplicidad), y aún no hemos estudiado sus hiperparámetros y si estos pueden hacer que mejore.

5.2. Random Forest

Es el turno de Random Forest, el modelo que mejores resultados ha obtenido en cuanto a precisión media y a varianza, que no tenía unas malas curvas de aprendi-

zaje, y que no es tan rápido como los SVM pero tienen una complejidad y tiempo de cómputo aceptable. Para intentar mejorar, o encontrar los hiperparámetros que mejor se ajustan a este modelo vamos a utilizar la misma técnica de antes, el *GridSearchCV*. En este caso, el grid y el modelo Random Forest se crea de la siguiente forma:

```
# Crear grid de hiperparametros que se van a probar
param_grid_rf = [{'n_estimators': np.linspace(100,500,9,dtype=np.int),
                  'max_depth': np.linspace(5, 15, 6, dtype=np.int),
                  'random_state': [1]}]

# Crear modelo de Random Forest
rf = RandomForestClassifier()
```

Para el hiperparámetro *n_estimators*, que determina el número de árboles, generamos una muestra de tamaño 9 con valores que van desde el 100 hasta el 500; y para *max_depth*, que determina la profundidad de los árboles, generaremos otra muestra, en este caso de tamaño 6 y con valores que van desde el 5 hasta el 15. El último hiperparámetro también será *random_state*, que al igual que antes, es la semilla para los aleatorios.

Como la técnica para realizar la búsqueda de los mejores valores es la misma que la utilizada anteriormente en Regresión Logística, no la vamos a volver a explicar, simplemente pasemos a ver los resultados que nos ha dado:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=13, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=150, n_jobs=None,
            oob_score=False, random_state=1, verbose=0, warm_start=False)

Mean training accuracy: 1.0
Training accuracy Std. Dev: 0.0
Mean CV accuracy: 0.9783549783549783
CV accuracy Std. Dev: 0.013591169253952318
```

Figura 2: Resultados de *GridSearch* sobre Random Forest.

Podemos ver que el mejor valor que ha encontrado para el *n_estimators* ha sido 150, es decir, más alto..... [esperar a confirmar]; mientras que para *max_depth* los mejores resultados se han dado con una profundidad de 13, por lo que es mejor... [esperar a confirmar].

6. Evaluación del mejor modelo

Referencias

[1] UCI. *Image segmentation*
http://archive.ics.uci.edu/ml/datasets/image+segmentation

[2] Scikit-Learn. *LogisticRegression*
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[3] Scikit-Learn. *SVC*
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

[4] Scikit-Learn. *RandomForestClassifier*
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier

[5] Scikit-Learn. *MLPClassifier*
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

[6] Scikit-Learn. *StandardScaler*
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[7] Scikit-Learn. *PCA*
https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[8] Scikit-Learn *plot_learning_curve*
https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr-auto-examples-model-selection-plot-learning-curve-py

[9] Scikit-Learn *GridSearchCV*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html