Práctica de aprendizaje supervisado

La opción elegida es la **opción 1** sobre la campaña de marketing del banco portugués. Para realizare esta práctica se ha utilizado un entorno de Python 3.5 y las librerías numpy y sklearn.

Clasificación binaria sobre la compra

El flujo completo se encuentra en *predice_compra.py*.

Metodología

Para elaborar un predictor sobre si un cliente realizará una compra o no, se han cargado en primera estancia todos los datos proporcionados (tanto variables categóricas como lineales). Para ello se ha usado la función *loadtxt* que proporciona numpy, usado los argumentos *delimiters* y *converters* para eliminar respectivamente los separadores y las comillas.

La mezcla (*np.random.shuffle*) se ha realizado por si el dataset no se hubiera muestreado de forma aleatoria, lo cual podría tener un efecto en el entrenamiento.

Las variables categóricas se han codificado usando el método *LabelBinarizer* que proporciona *sklearn*, lo cual nos facilita el paso de crear un diccionario de conversión de categorías a enteros, y de enteros a vectores *one-hot*. Se guarda las instancias debido a que si en un futuro se quisieran predecir muestras desconocidas, se deberán transformar los datos de la misma manera.

Separamos train y test set (no importa hacerlo tras la transformación de las variables categóricas, porque no transforma el significado de los datos). No separamos un set de validación porque vamos a usar *cross-validation* para la optimización de hiperparámetros.

Función loss

Para medir una estimación de lo bien que funcionan nuestros modelos se ha realizado una función personalizada. De partida, la clase destino está muy desproporcionada, de forma que hay muchos más casos de **no compra** que de **sí compra**. Es por ello que un estimador por defecto probablemente nos daría buena puntuación con predecirlo todo a **no compra**. Por ello, nuestro estimador mide la cantidad de fallos concretamente en los casos de **sí compra**.

```
def loss_func(y, y_pred):
    indices_yes = np.argwhere(y.ravel()=='yes').ravel()
    predichos_yes = y_pred[indices_yes]
    error = np.sum(predichos_yes=='no') / predichos_yes.shape[0]
    return np.log(1 + error)
```

Búsqueda de hiperparámetros

Cross-validation es un método que permite tunear hiperparámetros asociados a la selección de características así como de los modelos empleados, sin necesitar un set de validación. Funciona fragmentando el trainset en varias partes iguales, y obteniendo un *score* para cada fragmento habiendo sido entrenado el modelo con el resto de los fragmentos. En nuestro caso en particular, hemos fragmentado en 5 el trainset.

Para la búsqueda de hiperparámetros óptimos se ha utilizado búsqueda por gradilla (Grid Search) para cada uno de los modelos. Los modelos empleados han sido: **regresión logística, vector de máquina de soporte (SVC), perceptrón multicapa (MLP) y árbol de decisiones**.

Se ha probado cada uno de los modelos acoplado a un procesamiento previo de los datos (mediante un *pipeline*):

- Un filtrado de varianza (nos interesa que las variables independientes varíen, sino, no sirven para clasificar nada).
- Un análisis de componentes principales (PCA; lo cual permite reducir variables redundantes o linealmente separables, fusionándolas).
- SelectKBest, que selecciona las variables dependientes más correlacionadas a la variable dependiente.

Estos *pipelines* se han introducido como argumento de *GridSearchCV* para realizar el tuneado de hiperparámetros. Los hiperparámetros se han definido y pasado a la misma función con el argumento *param_grid*.

Los parámetros optimizados han sido:

- Umbral de varianza
- Número de clases tras PCA
- K-best
- ‘C’ (SVC)
- Kernel (SVC)
- Número de capas ocultas (MLP)
- Solver (MLP)
- max_iter (MLP)

Se han probado estos parámetros por separado y en conjunto en el dataset reducido (bank.csv), como se describe en los resultados.

Resultados

Procesamiento de datos

Utilizando el dataset reducido (de una décima parte de bank-full.csv), y tras desglosar las variables categóricas en sus respectivos vectores one-hot, los datasets quedaron así:

```
Shape del dataset (x): (4521, 16)
Shape del dataset (y): (4521,)
Shape del dataset codificado (x): (4521, 68)

Shape del trainset (x): (3390, 68)
Shape del trainset (y): (3390, 1)
Shape del testset (x): (1131, 68)
Shape del testset (y): (1131, 1)
```

Regresión logística

La regresión logística sin ningún tipo de ingeniería de *features*, da un porcentaje de éxito de entorno al 54%:

```
================================================================================
= Regresión logística
================================================================================
Score 0.5382577557744169 para params {}

Eficacia Reg. log. (test set): 54.02%
```

Tras descartar los datos poco variantes, que añaden ruido a un modelo que no sabe lidiar bien con él (como ése, la regresión logística), se comprueba que alcanzando un umbral de en torno a 0.25, se mejora la predicción hasta el 60% o un poco más.

```
================================================================================
= Regresión logística
================================================================================
Score 0.5635295260847413 para params {'filtra_varianza__threshold': 0.1}
Score 0.5751728900072164 para params {'filtra_varianza__threshold': 0.15}
Score 0.5808534404639165 para params {'filtra_varianza__threshold': 0.2}
Score 0.6003934032958538 para params {'filtra_varianza__threshold': 0.25}
Score 0.6003934032958538 para params {'filtra_varianza__threshold': 0.3}

Eficacia Reg. log. (test set): 62.27%
```

El análisis de componentes principales nos mejora aun más la predicción, alcanzando un óptimo con una sola variable en torno al 69%:

```
================================================================================
= Regresión logística
================================================================================
Score 0.6931471805599453 para params {'reduce_dim__n_components': 1}
Score 0.6051223608205002 para params {'reduce_dim__n_components': 5}
Score 0.5870355350559714 para params {'reduce_dim__n_components': 10}
Score 0.5656155116897452 para params {'reduce_dim__n_components': 30}
Score 0.5320142960431854 para params {'reduce_dim__n_components': 60}

Eficacia Reg. log. (test set): 69.31%
```

La selección de características por si sola también tiende a encontrar mejores resultados con una sola variable independiente, aunque peores que realizando un PCA.

```
================================================================================
= Regresión logística
================================================================================
Score 0.6091170712614596 para params {'selectkbest__k': 1}
Score 0.5823536787451259 para params {'selectkbest__k': 3}
Score 0.5794801155128158 para params {'selectkbest__k': 5}
Score 0.5749061943327121 para params {'selectkbest__k': 20}
Score 0.5704575999077168 para params {'selectkbest__k': 40}

Eficacia Reg. log. (test set): 61.31%
```

SVC

SVM es muy sensible a la magnitud de los datos, por lo que siempre añadimos una fase previa de normalización en el rango [-1,1].

La implementación SVC de SVM, con el kernel linear, y sin más procesamiento de datos, da una eficacia peor que la regresión logística con PCA.

```
================================================================================
= SVC
================================================================================
Score 0.6464306359688353 para params {'SVC__C': 0.01, 'SVC__kernel': 'linear'}

Eficacia SVC (test set): 63.06%
```

Aunque si combinamos este clasificador lineal con PCA, da un resultado muy parecido (~69%):

```
================================================================================
= SVC
================================================================================
Score 0.6931471805599453 para params {'SVC__C': 0.01, 'reduce_dim__n_components': 1, 'SVC__kernel': 'linear'}
Score 0.6931471805599453 para params {'SVC__C': 0.01, 'reduce_dim__n_components': 5, 'SVC__kernel': 'linear'}

Score 0.6931471805599453 para params {'SVC__C': 0.01, 'reduce_dim__n_components': 10, 'SVC__kernel': 'linear'}
Score 0.6931471805599453 para params {'SVC__C': 0.01, 'reduce_dim__n_components': 20, 'SVC__kernel': 'linear'}
Score 0.6931471805599453 para params {'SVC__C': 0.01, 'reduce_dim__n_components': 30, 'SVC__kernel': 'linear'}

Eficacia SVC (test set): 69.31%
```

Si comparamos la eficacia de SVC con un kernel que proyecta los datos a una dimensionalidad superior, entonces sin PCA y con el kernel RBF, se obtiene una eficacia similar (~69%):

```
================================================================================
= SVC
================================================================================
Score 0.6116793357578073 para params {'SVC__kernel': 'linear', 'SVC__C': 0.01}
Score 0.6931471805599453 para params {'SVC__kernel': 'rbf', 'SVC__C': 0.01}

Eficacia SVC (test set): 69.31%
```

SVC también es muy sensible en este caso al parámetro C, el cual por ejemplo es mucho mejor para C=0.01 que para C=10.

```
================================================================================
= SVC
================================================================================
Score 0.6931471805599453 para params {'SVC__C': 0.01, 'SVC__kernel': 'rbf'}
Score 0.5954086073781236 para params {'SVC__C': 10, 'SVC__kernel': 'rbf'}

Eficacia SVC (test set): 69.31%
```

Perceptrón multicapa

Los resultados siguientes nos revelan que la mejor eficacia se obtiene con el solver ‘LBFGS’, y con un número más bien pequeño de neuronas en nuestro modelo con una capa oculta. De nuevo, alcanzamos eficacia de en torno al 69% (sin haber usado PCA, k-best, ni filtro de varianza).

```
================================================================================
= MLP
================================================================================
Score 0.6880186944372777 para params {'MLP__max_iter': 1000, 'MLP__hidden_layer_sizes': (5,), 'MLP__solver':
'lbfgs'}
Score 0.6260114563445743 para params {'MLP__max_iter': 1000, 'MLP__hidden_layer_sizes': (5,), 'MLP__solver':
'adam'}
Score 0.6518238053399923 para params {'MLP__max_iter': 5000, 'MLP__hidden_layer_sizes': (5,), 'MLP__solver':
```

```
'lbfgs'}
Score 0.598270847771949 para params {'MLP__max_iter': 5000, 'MLP__hidden_layer_sizes': (5,), 'MLP__solver': 'adam'}
Score 0.6179046662949639 para params {'MLP__max_iter': 10000, 'MLP__hidden_layer_sizes': (5,), 'MLP__solver': 'lbfgs'}
Score 0.6146647795148734 para params {'MLP__max_iter': 10000, 'MLP__hidden_layer_sizes': (5,), 'MLP__solver': 'adam'}
Score 0.6261270835320032 para params {'MLP__max_iter': 1000, 'MLP__hidden_layer_sizes': (15,), 'MLP__solver': 'lbfgs'}
Score 0.6008754710062708 para params {'MLP__max_iter': 1000, 'MLP__hidden_layer_sizes': (15,), 'MLP__solver': 'adam'}
Score 0.6549850136261707 para params {'MLP__max_iter': 5000, 'MLP__hidden_layer_sizes': (15,), 'MLP__solver': 'lbfgs'}
Score 0.6288932296404071 para params {'MLP__max_iter': 5000, 'MLP__hidden_layer_sizes': (15,), 'MLP__solver': 'adam'}
Score 0.551813126406069 para params {'MLP__max_iter': 10000, 'MLP__hidden_layer_sizes': (15,), 'MLP__solver': 'lbfgs'}
Score 0.5755238483652313 para params {'MLP__max_iter': 10000, 'MLP__hidden_layer_sizes': (15,), 'MLP__solver': 'adam'}
Score 0.6301942121824653 para params {'MLP__max_iter': 1000, 'MLP__hidden_layer_sizes': (30,), 'MLP__solver': 'lbfgs'}
Score 0.5709586087008097 para params {'MLP__max_iter': 1000, 'MLP__hidden_layer_sizes': (30,), 'MLP__solver': 'adam'}
Score 0.5402460008116934 para params {'MLP__max_iter': 5000, 'MLP__hidden_layer_sizes': (30,), 'MLP__solver': 'lbfgs'}
Score 0.4906279332072881 para params {'MLP__max_iter': 5000, 'MLP__hidden_layer_sizes': (30,), 'MLP__solver': 'adam'}
Score 0.6345740886733547 para params {'MLP__max_iter': 10000, 'MLP__hidden_layer_sizes': (30,), 'MLP__solver': 'lbfgs'}
Score 0.5355938157508633 para params {'MLP__max_iter': 10000, 'MLP__hidden_layer_sizes': (30,), 'MLP__solver': 'adam'}


Eficacia MLP (test set): 69.31%
```

Árbol de decisiones

Si usamos SelectKBest previamente, parece que funciona de forma similar o quizá un poco mejor que la regresión logística en el test set.

```
================================================================================
= Árbol de decisiones
================================================================================
Score 0.5986599281084419 para params {'selectkbest__k': 1}
Score 0.5172824334913678 para params {'selectkbest__k': 3}
Score 0.5053178194076846 para params {'selectkbest__k': 5}


Eficacia árbol decisiones (test set): 64.44%
```

Los resultados con PCA son similares:

```
================================================================================
= Árbol de decisiones
================================================================================
Score 0.6281695334243906 para params {'reduce_dim__n_components': 1}
Score 0.5631440475730192 para params {'reduce_dim__n_components': 2}
Score 0.522434406152697 para params {'reduce_dim__n_components': 3}
Score 0.5012408364174361 para params {'reduce_dim__n_components': 4}

Eficacia árbol decisiones (test set): 62.2%
```

Interpretación de los datos y modelos

Los modelos de predicción lineales sin PCA previo muestran predecir con menos éxito que si se emplea PCA previo en el modelo lineal o bien un modelo que soporta dinámicas no lineales. Puesto que PCA no proyecta los datos a una dimensionalidad superior sino que simplemente reduce datos linealmente separables, esta mejora (hasta ~69% en el test set) no se debe a que en los datos subyazca una dinámica no lineal, sino a que tanto el uso de PCA como de uno de los modelos mencionados, son capaces de lidiar con el ruido que suponen las variables independientes redundantes o linealmente separables.

Regresión de la edad

Se han utilizado dos modelos de regresión para estimar la edad del cliente a partir de los demás parámetros: **regresión lineal**, y **perceptrón multicapa**. El código empleado está en el archivo predice_edad.py

Se ha usado R^2 para estimar la correlación, siendo peor cuanto más baja (0 es sin correlación), y mejor cuanto más cercana a 1 (completa correlación).

Los resultados de ambos modelos indican que hay una correlación R^2 de ~0,40 entre la edad y los demás parámetros, y que el modelo MLP es un poco mejor que el de regresión lineal (con PCA con 30 componentes, donde hay más eficacia tal y como se muestra).

```
================================================================================
= Regresión lineal
================================================================================
Score 0.003738596230808278 para params {'reduce_dim__n_components': 1}
Score 0.004430727160552838 para params {'reduce_dim__n_components': 5}
Score 0.13844704670446994 para params {'reduce_dim__n_components': 10}
Score 0.39056230774047035 para params {'reduce_dim__n_components': 30}
Score -5466115526599701.0 para params {'reduce_dim__n_components': 60}
R² sobre Reg. lin. (test set): 0.38
```

Se probaron distintas configuraciones para este regresor, y la mejor configuración que se obtuvo fue con ‘solver’ adam y 30 neuronas en la capa intermedia.

```
================================================================================
= MLP
================================================================================
Fitting 3 folds for each of 1 candidates, totalling 3 fits
[Parallel(n_jobs=3)]: Done   3 out of   3 | elapsed:    1.8s finished
Score 0.407116041155117 para params {'MLP__solver': 'adam', 'MLP__hidden_layer_sizes': (30,), 'MLP__max_iter': 5000}


R² sobre MLP (test set): 0.43
```