Proyecto Final: Online News Popularity

Aprendizaje Automático

Miguel Lentisco Ballesteros Antonio Coín Castro

Curso 2019-20

Índice

1. Introducción

En esta práctica perseguimos ajustar el mejor modelo lineal en dos conjuntos de datos dados, para resolver un problema de clasificación y otro de regresión. En ambos casos seguiremos una estrategia común que nos llevará finalmente a elegir un modelo y estimar su error.

1. Analizaremos la bases de datos y entenderemos el contexto del problema a resolver.
2. Preprocesaremos los datos de forma adecuada para trabajar con ellos.
3. Elegiremos una clase de hipótesis (lineal) para resolver el problema.
4. Fijaremos algunos modelos concretos y seleccionaremos el mejor según algún criterio.
5. Estimaremos el error del modelo.

Trabajamos en su mayoría con las funciones del paquete `scikit-learn`, apoyándonos cuando sea necesario en otras librerías como `numpy`, `matplotlib` ó `pandas`. El código de la práctica se ha estructurado en dos *scripts* de `Python` debidamente comentados:

- En `fit.py` se resuelve el problema de regresión.
- En `visualization.py` se recogen todas las funciones de visualización de gráficas, tanto comunes como propias de cada problema.

La ejecución de los dos programas principales está preparada para que se muestren solo algunas gráficas además del procedimiento de ajuste del modelo (aquellas que consumen menos tiempo). Este comportamiento puede cambiarse mediante el parámetro `show` de la función principal en cada caso, eliminando todas las gráficas (valor 0) o mostrándolas todas (valor 2). Además, en las operaciones de cálculo intensivo que lo permitan se utiliza el parámetro `n_jobs = -1` para paralelizar el flujo de trabajo en tantas hebras como se pueda. Por pantalla se muestra información sobre el tiempo de ejecución de los ajustes de los modelos y del programa completo.

Para que los experimentos sean reproducibles se fija una semilla aleatoria al incio del programa. Todos los resultados y gráficas que se muestran a continuación se han obtenido con el valor 2020 para la semilla, y pueden reproducirse si se ejecuta el programa tal y como se proporciona.

2. Base de datos y descripción del problema

Citar a (Fernandes, Vinagre, & Cortez, 2015) y poner link a la BBDD. Mencionar un resumen muy general de los diferentes atributos que se recogen (tipo real, entero, etc). Mostrar tabla-resumen que viene en el artículo.

2.1. Resumen estadístico de los datos (?)

Estudiar la distribución de clases si hacemos clasificación binaria (número exacto, porcentaje).

3. Selección de la clase de funciones

4. Conjuntos de entrenamiento, validación y *test*

5. Preprocesado de datos

5.1. Valores perdidos

5.2. Selección de características

5.3. Transformaciones polinómicas

5.4. Estandarización y umbral de varianza

5.5. Orden de las transformaciones

6. Métricas de error

7. Regularización

El uso de la regularización es esencial para limitar la complejidad del modelo y el *overfitting*, cosa que nos permitirá obtener una mejor capacidad de generalización. Consideramos la siguiente regularización:

- **Regularización L2 (Ridge)**: se añade una penalización a la función de pérdida que es cuadrática en los pesos,
$$L_{reg}(w) = L(w) + \lambda||w||_2^2.$$

El valor de $\lambda > 0$ es un hiperparámetro del modelo, que controla la intensidad de la regularización (a mayor valor, más pequeños serán los pesos). Encontrar un valor adecuado es una tarea complicada, pues si es demasiado pequeño seguiremos teniendo sobreajuste, pero si es demasiado grande podríamos caer en el fenómeno opuesto: tener *underfitting* porque el modelo sea poco flexible y no consiga ajustar bien los datos de entrenamiento.

Hemos considerado L2 ya que queremos generalizar los modelos para bajar la varianza en pos de extraer un mayor valor de la métrica, a costa de no rebajar el tiempo de computación (reduciendo variables con L1), pero no nos importa puesto que el tiempo de entrenamiento es razonable y preferimos aumentar el acierto.

8. Técnicas y selección de modelos

Pasamos a realizar la selección de hiperparámetros para cada modelo. Hemos considerado usar la técnica de K-fold cross validation para escoger la mejor configuración de hiperparámetros de cada modelo. Esta técnica se basa en dividir el conjunto de entrenamiento en K conjuntos de igual tamaño, y va iterando sobre ellos de forma que en cada paso entrena los modelos en los datos pertenecientes a $K-1$ de los conjuntos, y los evalúa en el conjunto restante. Finalmente se realiza la media del error a lo largo de todos los mini-conjuntos de validación y se escoge el

modelo con menor error. Este error se conoce como error de cross-validation, denotado E_{cv}, y sabemos que es un buen estimador del error fuera de la muestra.

Para esto utilizamos la función `GridSearchCV`, la cual puede recibir un pipeline como estimador y una lista de diccionarios que represente el espacio de hiperparámetros. Para evitar el fenómeno de data snooping que podría contaminar los conjuntos de validación, todo el cauce de preprocesado y selección de modelos se realiza de principio a fin: fijamos al principio las divisiones en K folds y las utilizamos en todo el proceso. Para el caso de clasificación, estas divisiones serán en particular instancias de `StratifiedKFold`, que respeta la distribución de clases en las divisiones.

Para todos los modelos, hacemos un preanálisis para estimar un buen espacio de búsqueda de los hiperparámetros, de manera que no es completamente arbitrario, si no que intentamos restringir a un entorno (holgado, puesto que usaremos menos datos) de la configuración óptima de hiperparámetros que hemos encontrado en este preanálisis. Además, al usar un conjunto menor nos permite explorar un espacio mucho más grande que usando el dataset de entrenamiento entero.

Una vez hemos encontrado la mejor configuración para cada modelo, se vuelve a entrenar sobre todo el conjunto de entrenamiento, para obtener un mejor rendimiento. Este es el comportamiento por defecto de la función `GridSearchCV`.

Comentamos ahora los modelos que pre-seleccionamos en el problema de clasificación. La métrica usada para decidir el mejor modelo será, de forma natural, el accuracy medio en los conjuntos de validación. Fijamos el número máximo de iteraciones en 1000 para todos los modelos que usen iteraciones.

8.1. Modelos lineales

8.1.1. Regresión Logística

En primer lugar consideramos un modelo de regresión logística, implementado en el objeto `LogisticRegression`, usando regularización L2. El parámetro de regularización, cuyo inverso es lo que en el código se alude como C, viene dado por el preanálisis, considerando 40 puntos en el espacio logarítmico $[-5, 1]$ para el preanálisis.

```
{"clf": [LogisticRegression(penalty = 'l2',
                            random_state = SEED,
                            max_iter = max_iter)],
 "clf__C": np.logspace(-5, 1, 40)}
```

En este caso, la técnica de optimización es la que viene por defecto, que se conoce como LBFGS. Se trata de un algoritmo iterativo similar al método de Newton para optimización, pero que utiliza una aproximación de la inversa de la matriz Hessiana. Se ha elegido porque tiene un tiempo de ejecución asumible y los resultados suelen ser buenos. La función a optimizar es la pérdida logarítmica:

$$L_{log}(w) = \frac{1}{N}\sum_{n=1}^{N} \log(1 + e^{-y_n w^T x_n}),$$

a la que se añade el término de penalización L2. Debemos tener en cuenta que aunque el algoritmo optimice esta función para proporcionar un resultado, la métrica de error que nosotros estamos usando es el accuracy y no el error logarítmico.

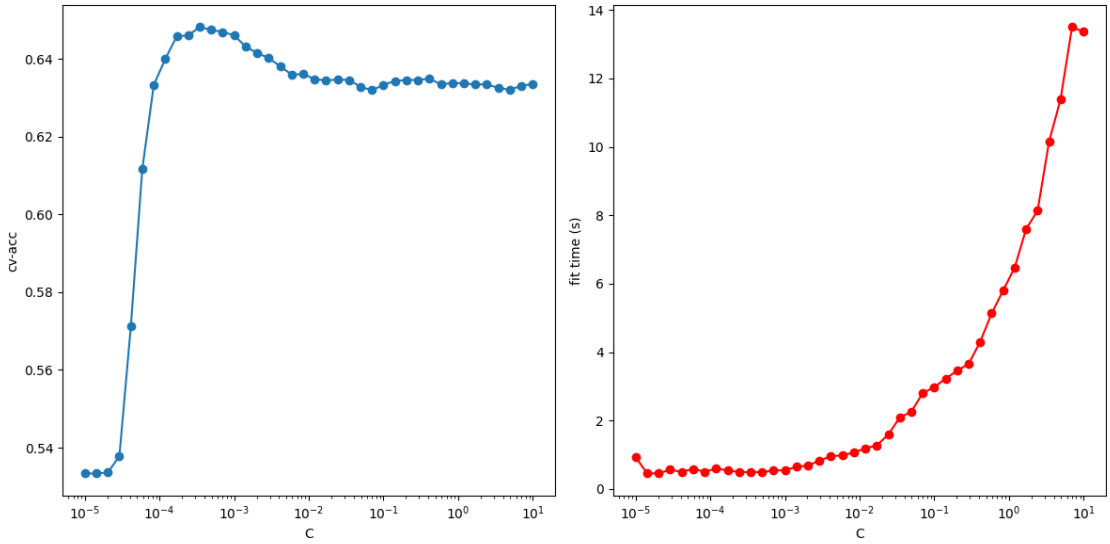

Figura 1: Análisis del hiperparámetro C de LogisticRegression.

El resultado del preanálisis de 1 nos indica que desde el orden de 10^{-4} en adelante, los resultados son igual de buenos, alcanzando un máximo entre 10^{-4} y 10^{-3}. Por tanto restringimos C al espacio logarítmico $[-4, 0]$, quedando:

```
{"clf": [LogisticRegression(penalty = 'l2',
                            random_state = SEED,
                            max_iter = max_iter)],
 "clf__C": np.logspace(-4, 0, 9)}
```

8.1.2. Regresión lineal

Consideramos también un modelo de regresión lineal. Utilizamos un objeto `RidgeClassifier`, que fija implícitamente la regularización L2. En este caso, la constante de regularización se llama alpha, considerando el espacio de búsqueda como 40 puntos en el espacio logarítmico $[-5, 5]$.

```
{"clf": [RidgeClassifier(random_state = SEED,
                         max_iter = max_iter)],
 "clf__alpha": np.logspace(-5, 5, 40)}
```

En este caso se pretende minimizar el error cuadrático de regresión (añadiendo regularización L2):

$$L_{lin}(w) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w^T x_n)^2.$$

Para ello se utiliza la técnica de la Pseudoinversa basada en la descomposición SVD de la matriz de datos, obteniendo una solución en forma cerrada y sin seguir un procedimiento iterativo. Se ha elegido esta técnica en lugar de SGD porque el tiempo de ejecución es más reducido y las soluciones obtenidas son suficientemente buenas.

El resultado del preanálisis de 2 nos arroja un máximo cerca de 10^4, por lo que restringimos el espacio de búsqueda al espacio logarítmico $[0, 5]$:

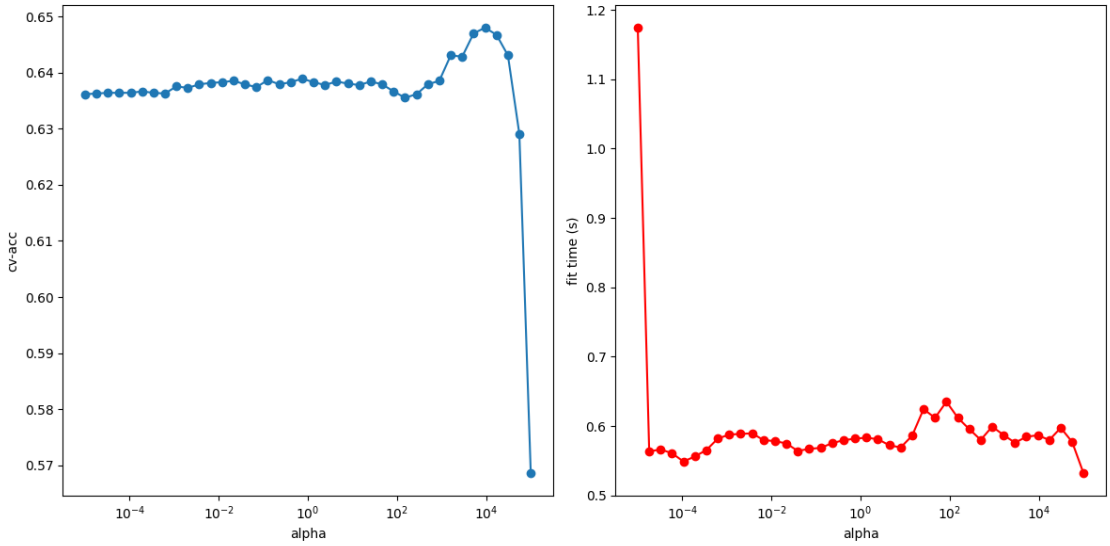

Figura 2: Análisis del hiperparámetro alpha de RidgeClassifier.

```
{"clf": [RidgeClassifier(random_state = SEED,
                         max_iter = max_iter)],
 "clf__alpha": np.logspace(0, 5, 9)}
```

8.1.3. SVM Lineal

Finalmente en modelos lineales, consideramos las máquinas de soporte vectorial (SVM) lineales (sin usar kernel) utilizando el objeto `SGDClassifier` con regularización L2, y tomando alpha con 40 puntos en el espacio logarítmico $[-6, 2]$.

```
{"clf": [SGDClassifier(random_state = SEED,
                       penalty = 'l2',
                       max_iter = max_iter)],
 "clf__alpha": np.logspace(-6, 2, 40)}
```

La técnica usada es SGD, de manera que minimizamos el error hinge junto con la regularización L2:

$$L_{svm}(w) = \frac{1}{n} \text{máx}(0, 1 - y_i f(x_i))$$

.

Los resultados del preanálisis en 3 nos indican un máximo cerca de 10^{-1} y 1, por lo que restringimos al espacio logarítmico $[-5, 1]$:

```
{"clf": [SGDClassifier(random_state = SEED,
                           penalty = 'l2',
                           max_iter = max_iter,
                           eta0 = 0.1)],
 "clf__learning_rate": ['optimal', 'invscaling', 'adaptive'],
 "clf__alpha": np.logspace(-5, 1, 7)}
```

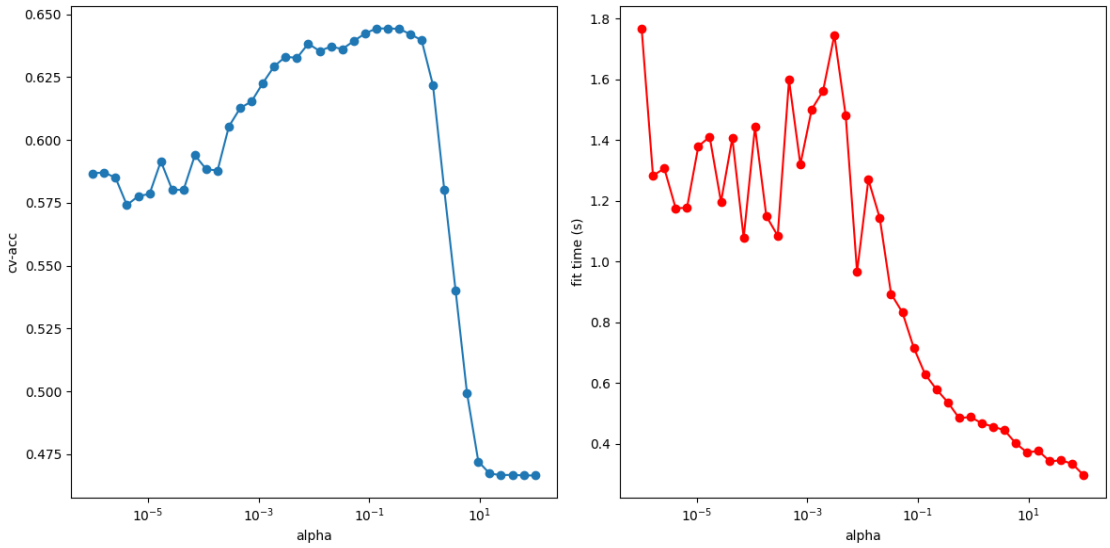

Figura 3: Análisis del hiperparámetro alpha de SGDClassifier.

También hemos considerado como hiperparámetro adicional el tipo de tasa de aprendizaje, fijando la inicial (`eta0`) como 0.1: `optimal` (tasa escogida por heurística), `adaptive` (inicial, y decrementa cuando el error no decrementa) y `invscaling` (inicial, y decrementa dividiendo por la raiz del nº de iteraciones).

8.2. Modelos no lineales

8.2.1. Random Forest (RF)

Consideramos primero

8.2.2. Boosting

8.2.3. Multilayer Perceptron (MLP)

8.2.4. K-Nearest Neighbors (KNN)

8.2.5. Funciones de Base Radial (RBF)

9. Análisis de resultados y estimación del error

10. Conclusiones y justificación

Anexo: Funcionamiento del código

Bibliografía

Fernandes, K., Vinagre, P., & Cortez, P. (2015). A proactive intelligent decision support system for predicting the popularity of online news. *Proceedings of the 17th portuguese conference on artificial intelligence*. Coimbra, Portugal.