Regularization

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]: %load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]: import warnings
   from sklearn.exceptions import DataConversionWarning
   warnings.filterwarnings(action='ignore', category=DataConversionWarning)
   warnings.filterwarnings(action='ignore', category=FutureWarning)
   warnings.filterwarnings(action='ignore', category=DeprecationWarning)

In [3]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [4]: from sklearn.compose import ColumnTransformer
   from sklearn.pipeline import Pipeline
   from sklearn.preprocessing import StandardScaler
   from sklearn.preprocessing import FunctionTransformer
   from sklearn.preprocessing import LabelEncoder
   from sklearn.tree import DecisionTreeClassifier
   from sklearn.linear_model import LinearRegression
   from sklearn.linear_model import LogisticRegression
   from sklearn.linear_model import Lasso
   from sklearn.linear_model import Ridge
   from sklearn.linear_model import SGDRegressor
   from sklearn.model_selection import cross_validate
   from sklearn.model_selection import validation_curve
   from sklearn.model_selection import cross_val_score
   from sklearn.model_selection import train_test_split

import matplotlib.gridspec as gridspec
In [5]: # Cork Property Prices Dataset
   # Use pandas to read the CSV file into a DataFrame
cork_df = pd.read_csv("datasets/dataset_corkA.csv")
# Shuffle
cork_df = cork_df.take(np.random.permutation(len(cork_df)))
# Get the target values
cork_y = cork_df["price"].values
# The features we want to select - we will only use the numeric features
cork_numeric_features = ["flarea", "bdrms", "bthrms", "floors"]

In [6]: # Create an OLS pipeline
cork_lr = Pipeline([
    ("pre", ColumnTransformer([(
        "num", FunctionTransformer(lambda x: x),
        cork_numeric_features)], remainder="drop")),
    ("est", LinearRegression())])

In [7]: # CS1109 Dataset
   # Use pandas to read the CSV file into a DataFrame
cs1109_df = pd.read_csv("datasets/dataset_cs1109.csv")
# Shuffle
cs1109_df = cs1109_df.take(np.random.permutation(len(cs1109_df)))
# Get the target values
cs1109_y = cs1109_df["outcome"].values
# The features we want to select
cs1109_features = ["lect", "lab", "cao"]

Introduction

- If your model underfits, we saw that one option is:
  - Stick with your existing model but remove constraints (if you can) to increase its complexity
- If your model overfits, we saw that one option is:
  - Stick with your existing model but add constraints (if you can) to reduce its complexity
- Constraining a model to make it less complex and reduce the risk of overfitting is called **regularization**

Regularization for Decision Trees

- Left unconstrained, the Decision Tree learning algorithm will learn a complicated tree that overfits the training data
- To avoid this, we can use regularization:
  - i.e. restrict the freedom of the learning algorithm
  - Most obvious is to restrict the depth the trees that it can learn

Example of regularization for Decision Trees

- Let's learn two Decision Trees on the CS1109 dataset:
  - one with no maximum depth
  - one with a maximum depth of 3
  - and compare their training accuracy and test accuracy
In [8]: # Create the Decision Tree pipelines

cs1109_dt_unconstrained = Pipeline([  
    ('pre', ColumnTransformer([  
        ('num', StandardScaler(), cs1109_features], remainder='drop')),  
    ('est', DecisionTreeClassifier())  
])

cs1109_dt_depth3 = Pipeline([  
    ('pre', ColumnTransformer([  
        ('num', StandardScaler(), cs1109_features], remainder='drop')),  
    ('est', DecisionTreeClassifier(max_depth=3))  
])

In [9]: # Accuracy Estimation

scores = cross_validate(cs1109_dt_unconstrained, cs1109_df, cs1109_y, cv=10, scoring="accuracy", return_train_score=True)
print("Training accuracy: ", np.mean(np.abs(scores["train_score"])))
print("Test accuracy: ", np.mean(np.abs(scores["test_score"])))

scores = cross_validate(cs1109_dt_depth3, cs1109_df, cs1109_y, cv=10, scoring="accuracy", return_train_score=True)
print("Training accuracy: ", np.mean(np.abs(scores["train_score"])))
print("Test accuracy: ", np.mean(np.abs(scores["test_score"])))

Training accuracy:  0.9866829006040291
Test accuracy:  0.695064935064935
Training accuracy:  0.8031034228308191
Test accuracy:  0.7419913419913421

• A classic case of overfitting! Explain!

Fitting Decision Trees with Different Amounts of Regularization

• We'll change the maximum depth from 1 to 29 inclusive
• We'll plot training accuracy and test accuracy

In [10]: depths = np.arange(1, 30)

cs1109_dt = Pipeline([  
    ('pre', ColumnTransformer([  
        ('num', StandardScaler(), cs1109_features], remainder='drop')),  
    ('est', DecisionTreeClassifier())  
])

accuracies_train, accuracies_test = validation_curve(  
    cs1109_dt, cs1109_df, cs1109_y, 'est__max_depth', depths, cv=10, scoring="accuracy")
mean_accuracies_train = np.mean(accuracies_train, axis=1)
mean_accuracies_test = np.mean(accuracies_test, axis=1)
In [11]:
    fig = plt.figure()
    plt.xlabel("max depth")
    plt.ylabel("Accuracy")
    plt.ylim(0.6, 1)
    plt.plot(depths, mean_accuracies_train, label = "training accuracy", color = "red")
    plt.plot(depths, mean_accuracies_test, label="test accuracy", color = "gold")
    plt.legend()
    plt.show()

    • A classic shape! Explain!

So how much regularization?

• In fact, reducing the maximum depth is not the only way to regularize Decision Tree learning
• You can increase the values of these hyperparameters: min_samples_split, min_samples_leaf, min_weight_fraction_leaf
• You can decrease the values of these hyperparameters: max_depth, max_leaf_nodes, max_features

Regularization for Linear Regression

• Linear models are among the least complex models
  ■ Hence, we normally associate them with underfitting
• But, even linear regression might overfit the training data
• If you are overfitting, you must reduce the degrees of freedom, e.g.
  ■ One way is to discard some features
     ▪ Then you have fewer coefficients \( \beta \) that you can modify
  ■ Another way is to constrain the range of values that the coefficients can take
     ▪ E.g. force the learning algorithm to only choose small values (close to zero)
     ▪ Recall that OLS linear regression finds coefficients \( \beta \) that minimize

\[
J(X, y, h_\beta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2
\]

Regularization imposes a penalty on the size of the coefficients
This is how we regularize linear regression (and logistic regression)
  ▪ In effect, it penalizes hypotheses that fit the data too well
Lasso Regression: Using the $l_1$-norm

- **Lasso Regression:**
  - ‘Lasso’ stands for ‘least absolute shrinkage and selection operator’ — but this doesn’t matter!
  - We penalize by the $l_1$-norm of $\beta$, which is simply the sum of their absolute values, i.e.
    \[ \sum_{j=1}^{n} |\beta_j| \]
  - (Minor point: we don’t penalize $\beta_0$, which is why $j$ starts at 1)
  - So Lasso Regression finds the $\beta$ that minimizes
    \[ J(X, y, h_\beta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} |\beta_j| \]
  - $\lambda$ is called the ‘regularization parameter’
    - It controls how much penalization we want and this determines the balance between the two parts of the modified loss function: fitting the data versus shrinking the parameters
    - As $\lambda \to 0$, Lasso Regression gets closer to being OLS Linear Regression
    - When $\lambda = 0$, Lasso Regression is the same as OLS Linear Regression
    - When $\lambda \to \infty$, penalties are so great that all the coefficients will tend to zero; the only way to minimize the loss function will be to make the coefficients as small as possible. It’s likely that in this case we will underfit the data.
  - So, for regularization to work well, we must choose the value of $\lambda$ carefully
    - So what kind of thing is $\lambda$?
  - **An important observation about Lasso Regression:**
    - As $\lambda$ grows, some of the $\beta_j$ will be driven to zero
    - This means that the model that it learns treats some features as irrelevant
    - Hence, it performs some **feature selection** too
    - Compare this with Ridge Regression below

**Implementing Lasso Regression**

- There is no equivalent to the Normal Equation
- Even Gradient Descent has a problem:
  - The Lasso loss function is not differentiable at $\beta_j = 0$
  - scikit-learn uses an approach called ‘coordinate descent’ (details unimportant)
    - There is a special class, Lasso
    - Or you can use SGDRegressor with penalty="l1"
  - They both refer to $\lambda$ as alpha!
  - Scaling is usually advised

```python
In [12]: # One approach:

cork_lasso = Pipeline([  
    ("pre", ColumnTransformer([(["num", StandardScaler()], cork_numeric_features),  
                                ("est", Lasso(alpha=1.0)))]))

In [13]: # Error estimation

np.mean(cross_val_score(cork_lasso, cork_df, cork_y, scoring="neg_mean_absolute_error", cv=10))
```

```
Out[13]: -61.605372225686075
```
Ridge Regression: Using the $l_2$-norm

- **Ridge Regression:**
  - We penalize by the $l_2$-norm, which is simply the sum of the squares of the coefficients, i.e.
    \[
    \sum_{j=1}^{n} \beta_j^2
    \]
    (Strictly speaking, the $l_2$-norm is the square root of the sum of squares)
  - So Ridge Regression finds the $\beta$ that minimizes
    \[
    J(X, y, h_\beta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \beta_j^2
    \]
  - Both Lasso and Ridge Regression shrink the values of the coefficients
    - But, as we mentioned, Lasso Regression may additionally result in coefficients being set to zero
    - This does not happen with Ridge Regression
    - Optionally, consult section 3.4.3 of *The Elements of Statistical Learning* by Hastie, Friedman & Tibshirani (available online) for an explanation
    - One observation from the book is that, roughly speaking, Lasso Regression shrinks the coefficients by approximately the same constant amount (unless they are so small that they get shrunk to zero), whereas, again roughly speaking, Ridge Regression shrinks the coefficients by approximately the same proportion
Implementing Ridge Regression

- There is an equivalent to the Normal Equation (solved, e.g., by Cholesky decomposition)
  - Take the gradient, set it equal to zero, and solve for $\beta$ (details unimportant):
    $$\beta = (X^T X + \lambda I)^{-1} X^T y$$
  - In the above, $I$ is the $(n + 1)$ identity matrix, i.e. all zeros except for the main diagonal which is all ones. (In fact, for consistency with what we were doing above, where we chose not to penalize $\beta_0$, you want a zero in the top left, so this is not really the identity matrix.)
  - Also, you don’t need to implement this with the pseudo-inverse. It’s possible to prove that, provided $\lambda > 0$, then $X^T X + \lambda I$ will be invertible
- Alternatively, use Gradient Descent:
  - The update rule for $\beta_j$ for all $j$ except $j = 0$ becomes:
    $$\beta_j \leftarrow \beta_j - \alpha \left( \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)} + \frac{\lambda}{m} \beta_j \right)$$
  - We can re-arrange this to:
    $$\beta_j \leftarrow \beta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)}$$
    which helps to show why this shrinks $\beta_j$
  - In scikit-learn, there is a special class, Ridge
    - You can set its solver parameter to choose different methods, or leave it as default auto
    - Or you can use SGDRegressor with penalty="l2"
- Scaling is usual advised

In [18]: # One approach (leaving solver="auto", so scikit-learn chooses which method to use)
    
cork_ridge = Pipeline([  
        ("pre", ColumnTransformer([(["num", StandardScaler(), cork_numeric_features]), remainder="drop"))),  
        ("est", Ridge(alpha=1.0))  
    ])

In [19]: # Error estimation
    
np.mean(cross_val_score(cork_ridge, cork_df, cork_y, scoring="neg_mean_absolute_error", cv=10))

Out[19]: -61.85234948089751

In [20]: # The other approach:
    
cork_ridge = Pipeline([  
        ("pre", ColumnTransformer([(["num", StandardScaler(), cork_numeric_features]), remainder="drop"))),  
        ("est", SGDRegressor(penalty="l2", alpha=1.0, max_iter=1500))  
    ])

In [21]: # Error estimation
    
np.mean(cross_val_score(cork_ridge, cork_df, cork_y, scoring="neg_mean_absolute_error", cv=10))

Out[21]: -78.0630207710212

# The result should be the same as using Ridge. The fact that it isn't suggests that I haven't set # the hyperparameters identically
Illustrating the Effects of Lasso and Ridge Regression

- We'll generate a random, non-linear dataset
- Then we'll fit an unregularized linear model and two regularized models (Lasso and Ridge)
- You can play with the regularization hyperparameter ($\lambda$, but called $\alpha$) to see how it affects the fit

```python
In [22]:
    def make_dataset(m, func, error):
        X = np.random.random(m)
        y = func(X, error)
        return X.reshape(m, 1), y

    def f(x, error=1.0):
        y = 10 - 1 / (x + 0.1)
        if error > 0:
            y = np.random.normal(y, error)
        return y

In [23]: X, y = make_dataset(50, f, 1.0)

In [24]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=np.random)
      ols = LinearRegression()
      ols.fit(X_train, y_train)
      y_predicted_ols = ols.predict(X_test)
      mse_ols = mean_squared_error(y_predicted_ols, y_test)

      lasso = Lasso(alpha=1.0)
      lasso.fit(X_train, y_train)
      y_predicted_lasso = lasso.predict(X_test)
      mse_lasso = mean_squared_error(y_predicted_lasso, y_test)

      ridge = Ridge(alpha=1.0)
      ridge.fit(X_train, y_train)
      y_predicted_ridge = ridge.predict(X_test)
      mse_ridge = mean_squared_error(y_predicted_ridge, y_test)
```
In [25]:
    # Set up the two subplots
    fig = plt.figure(figsize=(14, 4.5))
    gs = gridspec.GridSpec(1, 3)
    # Leftmost diagram: OLS
    ax0 = plt.subplot(gs[0])
    plt.title("OLS Linear Regression\nMSE: %.3f\nIntercept: %.3f\nCoefficient: %.3f" % (mse_ols, ols.intercept_, ols.coef_[0]))
    plt.xlabel("Feature")
    plt.ylabel("MSE")
    plt.ylim(-4, 14)
    ax0.scatter(X_train, y_train, color = "green")
    ax0.plot(X_test, y_predicted_ols, color = "blue")
    # Middle diagram: Lasso
    ax1 = plt.subplot(gs[1])
    plt.title("Lasso Regression\nMSE: %.3f\nIntercept: %.3f\nCoefficient: %.3f" % (mse_lasso, lasso.intercept_, lasso.coef_[0]))
    plt.xlabel("Feature")
    plt.ylabel("MSE")
    plt.ylim(-4, 14)
    ax1.scatter(X_train, y_train, color = "green")
    ax1.plot(X_test, y_predicted_lasso, color = "blue")
    # Rightmost diagram: Ridge
    ax2 = plt.subplot(gs[2])
    plt.title("Ridge Regression\nMSE: %.3f\nIntercept: %.3f\nCoefficient: %.3f" % (mse_ridge, ridge.intercept_, ridge.coef_[0]))
    plt.xlabel("Feature")
    plt.ylabel("MSE")
    plt.ylim(-4, 14)
    ax2.scatter(X_train, y_train, color = "green")
    ax2.plot(X_test, y_predicted_ridge, color = "blue")
    fig.tight_layout()
    plt.show()

Concluding Remarks

- For completeness, we mention Elastic Net, which combines Lasso and Ridge regularization, with yet another hyperparameter to control the balance between the two
- Using some regularization is usually better than none, and Ridge is a good default
- But we now have an extra hyperparameter whose value we must choose

In [ ]: