stat4500notes

Yi Wang

2023-09-19

Table of contents

Preface

This is a Lecture note written for the course STAT 4500: Machine Learning offered at Auburn University at Montgomery. The course uses the textbook James et al. (2023).

This is a book wrtieen by Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Setting up Python Computing Environment

1.1 on Your own computer

1. you can either `git clone` or download a zipped file containing the codes from the site: https://github.com/intro-stat-learning/ISLP_labs/tree/stable. If downloaded a zipped file of the codes, unzipped the file to a folder, for example, named `islp`. If `git clone` (preferred, you need to have Git installed on your computer, check this link for how to install `Git` https://ywanglab.github.io/stat1010/git.html), do `git clone https://github.com/intro-stat-learning/ISLP_labs.git`
2. Download and install the following software:
   - **Anaconda**: Download anaconda and install using default installation options
   - **Visual Studio Code** (VSC): Download VSC and install
   - start VSC and install VSC extensions in VSC: Python, Jupyter, intellicode
   - (optional) **Quarto** for authoring: Download Quarto and install
3. Create a virtual environment named `islp` for Python. Start an anaconda terminal.

   ```
   conda create -n islp python==3.10
   conda activate islp
   conda install pip ipykernel
   pip install -r https://raw.githubusercontent.com/intro-stat-learning/ISLP_labs/v2.1.2/
   ```

4. You are ready to run the codes using `VSC` or `jupyter lab`.
   - Activate the venv: `conda activate islp`
   - Start a Anaconda terminal, navigate to the folder using the command `cd path/to/islp`, where `path/to/islp` means the file path to the folder `islp`, such as `\Users\ywang2\islp`. Start VSC by typing `code .` in the anaconda terminal.
   - open/create a `.ipynb` or `.py` file.
   - Select the kernel `islp`
   - Run a code cell by pressing `Shift+Enter` or click the triangular play button.

- Continue to run other cells.
- After finishing using VSC, close the VSC, and deactivate the virtual environment in a conda terminal: `conda deactivate`

1.2 Use Google Colab

All you need is a Google account. Sign in your Google account in a browser, and navigate to Google Colab. Google Colab supports both `Python` and `R`. `Python` is the default engine. Change the engine to `R` in `Connect->change runtime type`. Then you are all set. Your file will be saved to your Google Drive or you can choose to send it to your `GitHub` account (recommended).

1.2.1 How to run a project file from your Google Drive?

Many times, when you run a python file in Colab, it needs to access other files, such as data files in a subdirectory. In this case, it would be convenient to have the same file structure in the Google Colab user home directory. To do this, you can use Google Drive to store your project folder, and then mount the Google Drive in Colab.

Let's assume the project folder name, `islp/`.Here are the steps:

1. `git clone` the project folder (example: `git clone https://github.com/intro-stat-learning/ISLP_l` to your local folder. This step is only needed when you want to clone some remote repo from GitHub.
2. **Upload** the folder (ex: `islp`) to Google Drive.
3. **Open the file using Colab**. In Google Drive, double click on the `ipynb` file, example, `ch06.ipynb` (or click on the three dots on the right end, and choose `open with`, then `Google Colaborotary`), the file will be opened by Google Colab.
4. **Mount the Google Drive**. In Google Colab, with the specific file (example, `ch06.ipynb`) being opened, move your cursor to the first code cell, and then click on the folder icon (this should be the fourth icon) on the upper left border in the Colab browser. This will open the file explorer pane. Typically you would see a folder named `sample_data` shown. On the top of the pane, click on the Google Drive icon to mount the Google Drive. Google Colab will insert the following code below the cursor in your opened `ipynb` file:

   ```
   from google.colab import drive
   drive.mount('/content/drive')
   ```

Run this code cell by pressing `SHIFT+ENTER`, and follow the prompts to complete the authentication. Wait for ~10 seconds, your Google Drive will be mounted in Colab, and it will be displayed as a folder named `drive` in the file explorer pane. You might need to click on the `Refresh` folder icon to see the folder `drive`.

5. Open a new code cell below the above code cell, and type the code

   ```
   %cd /content/drive/MyDrive/islp/
   ```

   This is to change the directory to the project directory on the Google Drive. Run this code cell, and you are ready to run the file `ch06.ipynb` from the folder `islp` on your personal Google Drive, just like it's on your local computer.

2 Chapter 2: Statistical Learning

2.1 What is statistical learning?

For the input variable $X \in \mathbb{R}^p$ and response variable $Y \in \mathbb{R}$, assume that

$$Y = f(X) + \epsilon,$$

where ϵ is a random variable representing **irreducible error**. We assume ϵ is *independent* of X and $E[\epsilon] = 0$. ϵ may include *unmeasured variables* or *unmeasurable variation*.

Statistical learning is to estimate f using various methods. Denote the estimate by $\hat{f}$.

- regression problem: when Y is a continuous (quantitative) variable . In this case $f(x) = E(Y|X = x)$ is the population *regression* function, that is, regression finds a conditional expectation of Y.
- classification problem: when Y only takes small number of discrete values, i.e., qualitative (categorical).

Logistic regression is a classification problem, but since it estimates class probability, it may be considered as a regression problem.

- supervised learning: training data $\mathcal{T}r = \{(x_i, y_i) : i \in \mathbb{Z}_n\}$: linear regression, logistic regression
- unsupervised learning: when only x_i are available. clustering analysis, PCA
- semi-supervised learning: some data with labels (y_i), some do not.
- reinforcement learning: learn a state-action policy function for an agent to interacting with an environment to maximize a reward function.

2.2 Why estimate f?

We can use estimated $\hat{f}$ to

- make predictions for a new X,
$$\hat{Y} = \hat{f}(X).$$

The prediction error may be quantified as

$$E[(Y - \hat{Y})^2] = (f(X) - \hat{f})^2 + \text{Var}[\epsilon].$$

The first term of the error is *reducible* by trying to improve $\hat{f}$, where we assume f, $\hat{f}$ and X are fixed.

- make inference, such as
  - Which predictors are associated with the response?
  - what is the relationship between the response and each predictor?
  - is the assumed relationship adequate? (linear or more complicated?)

2.3 How to estimate f

We use obtained observations called **training data** $\{(x_k, y_k) : k \in \mathbb{Z}_n\}$ to train an algorithm to obtain the estimate $\hat{f}$.

- Parametric methods: first assume there is a function form (shape) with some parameters. For example, a linear regression model with two parameters. Then use the *training data* to **train** or **fit** the model to determine the values of the parameters.

  Advantages: simplify the problem of fit an arbitrary function to estimate a set of parameters.

  Disadvantages: may not be flexible unless with large number of parameters and/or complex function shapes.

  Example: linear regression,

- Non-parametric methods: Do not explicitly assume a function form of f. They seek to estimate f directly using data points, can be quite flexible and accurate.

  **Disadvantage: need large number of data points

  Example: KNN (but breakdown for higher dimention. Typically only for $p \leq 4$), spline fit.

2.4 How to assess model accuracy

For regression problems, the most commonly used measure is the *mean squared error* (MSE), given by

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

For classification problems, typically the following **error rate** (classifications error) is calculated:

$$\frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i)$$

The accuracy on a training set can be arbitrarily increased by increasing the model flexibility. However, we are in general interested in the error on the test set rather on the training set, the model accuracy should be assessed on a test set.

Flexible models tend to overfit the data, which essentially means they follow the error or *noise* too closely in the training set, therefore cannot be generalized to *unseen cases* (test set).

2.5 Model Selection:

No free lunch theorem

There is no single best method for all data sets, which means some method works better than other methods for a particular dataset. Therefore, one needs to perform model selections. Here are some principles.

2.5.1 Trade-off between Model flexibility and Model Interpretability

More flexible models have higher *degree of freedom* and are less interpretable because it's difficult to interpret the relationship between a predictor and the response.

LASSO is less flexible than linear regression. GAM (generalized additive model) allows some non-linearity. Full non-linear models have higher flexibility, such as *bagging, boosting, SVM*, etc.

When *inference* is the goal, then there are advantages to using simple and less flexible models for interpretability.

When *prediction* is the main goal, more flexible model may be a choice. But sometimes, we obtain more accurate prediction using a simpler model because the underlying dataset has a simpler structure. Therefore, it is not necessarily true that a more flexible model has a higher prediction accuracy.

Occam's Razor: Among competing hypotheses that perform equally well, the one with the fewest assumptions should be selected.

2.5.2 Model Selection: the Bias-Variance Trade-off

As the model flexibility increases, the training MSE (or error rate for classificiton) will decrease, but the test MSE (error rate) in general will not and will show a characteristic **U-shape**. This is because when evaluated at a test point x_0, the expected test MSE can be decomposed into

$$E\left[(y_0 - \hat{f}(x_0))^2\right] = \text{Var}[\hat{f}(x_0)] + (\text{Bias}(\hat{f}(x_0)))^2 + \text{Var}[\epsilon]$$

where the expectation is over different $\hat{f}$ on a different training set or on a different training step if the training process is stochastic, and

$$\text{Bias}(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0)$$

To obtain the least test MSE, one must trade off between variance and bias. Less flexible model tendes to have higher bias, and more flexible models tend to have higher variance. An optimal flexibility for the least test MSE varies with different data sets. Non-linear data tends to require higher optimal flexibility.

2.6 Bayes Classifier

It can be shown that Bayes Classifier minimizes the classification test error

$$\text{Ave}(I(y_0 \neq \hat{y}_0)).$$

A Bayes Classifier assigns a test observation with predictor x_0 to the class for which

$$\Pr(Y = j|X = x_0)$$

is largest. It's error rate is given by

$$1 - E[\max_j \Pr(Y = j|X)]$$

where the expectation is over X. The Bayes error is analogous to the irreducible error ϵ.

Bayes Classifier is not attainable as we do not know $\Pr(Y|X)$. We only can estimate $\Pr(Y|X)$. One way to do this is by KNN. KNN estimate the conditional probability simply with a majority vote. The flexibility of KNN increases as $1/K$ increases with $K = 1$ being the most flexible KNN. The training error is 0 for $K = 1$. A suitable K should be chosen for an appropriate trade off between bias and variance. The KNN classifier will classify the test point x_0 based on the probability calculated from the k nearest points. KNN regression on the other hand will assign the test point x_0 the average value of the k nearest neighbors.

2.7 Homework (* indicates optional):

- Conceptual: 1,2,3,4*,5,6,7
- Applied: 8, 9*, 10*

2.8 Code Gist

2.8.1 OS

```
import os
os.chdir(path) # change dir
```

2.8.2 Python:

Concatenation using `+`

```
"hello" + " " + "world"  # 'hello world'
[3,4,5] + [4,9,7] # [3,4,5, 4,9,7]
```

String formatting using `string.format()`

```
print('Total is: {0}'.format(total))
```

`zip` to loop over a sequence of tuples

```
for value, weight in zip([2,3,19],
                         [0.2,0.3,0.5]):
    total += weight * value
```

2.8.3 Numpy

2.8.3.1 Numpy functions:

`np.sum(x)`, `np.sqrt(x)` (entry wise). `x**2` (entry wise power), `np.corrcoef(x,y)` (find the correlation coefficient of array `x` and array `y`)

`np.mean(axis=None)`: axis could be `None` (all entries), `0`(along row), `1`(along column)

`np.var(x, ddof=0)`, `np.std(x, ddof=0)`, # Note both `np.var` and `np.std` accepts an argument `ddof`, the divisor is `N-ddof`.

`np.linspace(-np.pi, np.pi, 50)` # start, end, number of points 50

`np.multiply.outer(row,col)` # calculate the product over the mesh with vectors `row` and `col`.

`np.zeros(shape or int, dtype)` #eg: `np.zeros(5,bool)`

`np.ones(Boston.shape[0])`

`np.all(x)`, `np.any(x)`: check if all or any entry of `x` is true.

`np.unique(x)`: find unique values in `x`. `np.isnan(x)`: return a boolean array of `len(x)`. `np.isnan(x).mean()`: find the percentage of `np.nan` values in `x`.

2.8.3.2 Array Slicing and indexing

`np.arange`(start, stop, step) # numpy version of `range`

`x[slice(3:6)]` # equivalent to `x[3:6]`

Indexing an array using `[row, col]` format. If `col` is missing, then index the entire rows. `len(row)` must be equal to `len(col)`. Otherwise use iterative indexing or use `np.ix_(x_idx, y_idx)` function, or use Boolean indexing, see below.

```
A[1,2]: index entry at row 1 and col 2 (recall Python index start from 0)
A[[1,3]] # row 1 and 3. Note the outer [] is considered as the operator, so only row indices
A[:,[0,2]] # cols 0 and 2
A[[1,3], [0,2,3]] # entry A[1,0] and A[3,2]
A[1:4:2, 0:3:2] # entries in rows 1 and 3, cols 0 and 2
A[[1,3], [0,2,3]] # syntax error
# instead one can use the following two methods
A[[1,3]][:,[0,2]] # iterative subsetting
A[np.ix_([1,3],[0,2,3])] # use .ix_ function to create an index mesh
A[keep_rows, keep_cols] # keep_rows, keep_cols are boolean arrays of the same length of rows
A[np.ix_([1,3],keep_cols)] # np.ix_()can be applied to mixture of integer array and boolean a
```

2.8.3.3 Random numbers and generators

```
np.random.normal(loc=0.0, scale=1.0,size=None) # size can be an integer or a tuple.
#
rng = np.random.default_rng(1303) # set random generator seed
rng.normal(loc=0, scale=5, size=2) #
rng.standard_normal(10) # standard normal distribution of size 10
rng.choice([0, np.nan], p=[0.8,0.2], size=A.shape)
```

2.8.3.4 Numpy array atributes

`.dtype`, `.ndim`, `.shape`

2.8.3.5 Numpy array methods

`x.sum(axis=None)` (equivalent to `np.sum(x)`), `x.T` (transpose),

`x.reshape((2,3))` # x.reshape() is a reference to x.

`x.min()`, `x.max()`

2.8.4 Graphics

2.8.4.1 2-D figure

```
# Using the subplots + ax methods
fig, ax = subplots(nrows=2, ncols=3, figsize=(8, 8))
# explicitly name each axis in the grid
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, figsize=(10,10))

ax[0,1].plot(x, y,marker='o', 'r--', linewidth=3); #line plot. `;` suppresses the text output
ax.plot([min(fitted),max(fitted)],[0,0],color = 'k',linestyle = ':', alpha = .3)
ax.scatter(x, y, marker='o'); #scatter plot
ax.scatter(fitted, residuals, edgecolors = 'k', facecolors = 'none')
ax.set_xlabel("this is the x-axis")
ax.set_ylabel("this is the y-axis")
ax.set_title("Plot of X vs Y");
axes[0,1].set_xlim([-1,1]) # set x_lim. similarly `set_ylim()`

fig = ax.figure  # get the figure object from an axes object
fig.set_size_inches(12,3) # access the fig object to change fig size (width, height)
fig # re-render the figure
```

```
fig.savefig("Figure.pdf", dpi=200); #save a figure into pdf. Other formats: .jpg, .png, etc
```

2.8.4.2 Contour and image

```
fig, ax = subplots(figsize=(8, 8))
x = np.linspace(-np.pi, np.pi, 50)
y = x
f = np.multiply.outer(np.cos(y), 1 / (1 + x**2))
ax.contour(x, y, f, levels=None); # numbre of levels. if None, automatically choose
ax.imshow(f); # heatmap colorcoded by f
```

2.8.5 Pandas

2.8.5.1 loading data

```
pd.read_csv('Auto.csv') # read csv
pd.read_csv('Auto.data',
            na_values =['?'], #specifying the na_values in the datafile.
            delim_whitespace=True) # read whitespaced text file
pd.read_csv('College.csv', index_col=0) # use column `0` as the row labels
```

2.8.5.2 Pandas Dataframe attributes and methods

```
Auto.shape
Auto.columns # gets the list of column names
Auto.index #return the index (labels) objects
Auto['horsepower'].to_numpy() # convert to numpy array
Auto['horsepower'].sum()

Auto.dropna() # drop the rows containing na values.
df.drop('B', axis=1, inplace=True) # drop a column 'B' inplace.
#equivalent to df.drop(columns=['B'], inplace=True)
df.drop(index=['Ohio','Colorado']) #eqivalent to: df.drop(['Ohio','Colorado'], axis=0)
auto_df.drop(auto_df.index[10:86]) # drop rows with index[10:86] not including 86

Auto.set_index('name')# rename the index using the column 'name'.

pd.Series(Auto.cylinders, dtype='category') # convert the column `cylinders` to 'category` dt
# the convertison can be done using `astype()` method
Auto.cylinders.astype('category')
Auto.describe() # statistics summary of all columns
```

```
Auto['mpg'].describe() # for selected columns

college.rename({'Unnamed: 0': 'College'}, axis=1): # change column name,
# alternavie way
college_df.rename(columns={college_df.columns[0] : "College"}, inplace=True) #

college['Elite'] = pd.cut(college['Top10perc'],  # binning a column
                          [0,0.5,1],  #bin edges
                          labels=['No', 'Yes'],  # bin labels (names)
                          right=True,# True: right-inclusive (default) for each bin ( ]; Fals
                          )
college['Elite'].value_counts() # frequency counts
auto.columns.tolist() # equivalent to  auto.columns.format() (rarely used)
```

2.8.5.3 Selecting rows and columns

Select Rows:

```
Auto[:3] # the first 3 rows.
Auto[Auto['year'] > 80] # select rows with boolean array
Auto_re.loc[['amc rebel sst', 'ford torino']] #label_based row selection
Auto_re.iloc[[3,4]] #integer-based row seleciton: rows 3 and 4 (index starting from 0)
```

Select Columns

```
Auto['horsepower'] # select the column 'horsepower', resulting a pd.Series.
Auto[['horsepower']] #obtain a dataframe of the column 'horsepower'.
Auto_re.iloc[:,[0,2,3]] # intger-based selection
auto_df.select_dtypes(include=['int16','int32']) # select columns by dtype
```

Select a subset

```
Auto_re.iloc[[3,4],[0,2,3]] # integer-based
Auto_re.loc['ford galaxie 500', ['mpg', 'origin']] #label-based
Auto_re.loc[Auto_re['year'] > 80, ['weight', 'origin']] # mix bolean indexing with labels

Auto_re.loc[lambda df: (df['year'] > 80) & (df['mpg'] > 30),
            ['weight', 'origin']
           ]  # using labmda function with loc[]
```

2.8.5.4 Pandas graphics

Without using `subplots` to get axes and figure objects

```
ax = Auto.plot.scatter('horsepower', 'mpg') #scatter plot of 'horsepower' vs 'mpg' from the c
ax.set_title('Horsepower vs. MPG');
fig = ax.figure
fig.savefig('horsepower_mpg.png');

plt.gcf().subplots_adjust(bottom=0.05, left=0.1, top=0.95, right=0.95) #in percentage of the
ax1.fig.suptitle('College Scatter Matrix', fontsize=35)
```

Using `subplots`

```
fig, axes = subplots(  ncols=3, figsize=(15, 5))
Auto.plot.scatter('horsepower', 'mpg', ax=axes[1]);
Auto.hist('mpg', ax=ax);
Auto.hist('mpg', color='red', bins=12, ax=ax); # more customized
```

Boxplot using `subplots`

```
Auto.cylinders = pd.Series(Auto.cylinders, dtype='category') # needs to convert the `cylinder
fig, ax = subplots(figsize=(8, 8))
Auto.boxplot('mpg', by='cylinders', ax=ax);
```

Scatter matrix

```
pd.plotting.scatter_matrix(Auto); # all columns
pd.plotting.scatter_matrix(Auto[['mpg',
                                 'displacement',
                                 'weight']]);  # selected columns


#Alternatively with sns.pairplot
```

Sns Graphic

```
# Scatter matrix
ax1 = sns.pairplot(college_df[college_df.columns[0:11]])

# Boxplot
sns.boxplot(ax=ax, x="Private", y="Outstate", data=college_df)
```

3 Chapter 3: Linear Regression

Linear regression is a simple supervised learning assuming a linear relation between Y and X. When there is only one predictor, it's a **simple linear regression**. When there are more than one predictors, it's called **multiple linear regression**. Note *multivariate regression* refer to the Y variable is a vector.

3.1 Simple Linear Regression

Assumes the *population regression line* model

$$Y = \beta_0 + \beta_1 X + \epsilon,$$

where, β_0 is the *expected* value of Y when $X = 0$, and β_1 is the *average* change in Y with a one-unit increase in X. ϵ is a "catch all" error term.

After training using the training data, we can obtain the parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$. The we can obtain the prediction for x given by the *least square line*:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

The error at a data point x_i is given by $e_i = y_i - \hat{y}_i$, and the *residual sum of squares* (RSS) is

$$\text{RSS} = e_1^2 + \cdots + e_n^2.$$

One can use the least square approach to minimize RSS to obtain

$$\hat{\beta}_1 = \frac{(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} = r_{xy}\frac{\sigma_y}{\sigma_x}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

where, $\bar{y} = \frac{1}{n}\sum_{i=1}^n y_i$ and $\bar{x} = \frac{1}{n}\sum_{i=1}^n x_i$. If we assume the data matrix X is demeaned, then $\hat{beta}_0 = \bar{y}.$ and the correlation

$$r_{xy} = \frac{\text{cov}(x,y)}{\sigma_x \sigma_y} = \frac{(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}. \tag{3.1}$$

is the normalized covariance. Note $-1 \leq r_{xy} \leq 1$. When there is no intercept, that is $\beta_0 = 0$, then

$$\hat{y}_i = x_i\hat{\beta} = \sum_{i=1}^{n} a_i y_i$$

where,

$$\hat{\beta} = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2}$$

That is, the fitted values are linear combinations of the response values when there is no intercept.

3.1.1 Assessing the accuracy of the coefficients

Let $\sigma^2 = \text{Var}(\epsilon)$, that is, σ^2 is the variance of Y, (estimated by $\sigma^2 \approx= \text{RSE} = \text{RSS}/(n-p-1)$.) Assume each observation have *common variance* (homoscedasticity) and are *uncorrelated*, then the standard errors under repeated sampling

$$(\text{SE}[\hat{\beta}_1])^2 = \frac{1}{\sigma_x^2} \cdot \frac{\sigma^2}{n}$$

$$(\text{SE}[\hat{\beta}_0])^2 = \left[1 + \frac{\bar{x}^2}{\sigma_x^2}\right] \cdot \frac{\sigma^2}{n}$$

- when x_i are more spread out (with large σ_x^2), then $\text{SE}[\hat{\beta}_1]$ is small. This is because there are more *leverage* (of x values) to estimate the slope.
- when $\bar{x} = 0$, then $\text{SE}[\hat{\beta}_0] = \text{SE}[\bar{y}]$. In this case, $\hat{\beta}_0 = \bar{y}$.

Standard errors are used to construct CI and perform hypothesis test for the estimated $\hat{\beta}_0$ or $\hat{\beta}_1$. Under the assumption of **Gaussian error**, One can construct the CI of significance level α (e.g., $\alpha = 0.05$) as

$$\hat{\beta}_j = [\hat{\beta}_j - t_{1-\alpha/2,n-p-1} \cdot \text{SE}[\hat{\beta}_j], \hat{\beta}_j + t_{1-\alpha/2,n-p-1} \cdot \text{SE}[\hat{\beta}_j]]$$

Where $j = 0, 1$. Large interval including zero indicates β_j is not statistically significant from 0. When n is sufficient large, $t_{0.975,n-p-1} \approx 2$. With the standard errors of the coefficients, one can also perform **hypothesis test** on the coefficients. For $j = 0, 1$,

$$H_0 : \beta_j = 0$$
$$H_A : \beta_j \neq 0$$

The t-statistic of degree $n-p-1$, given by

$$t = \frac{\hat{\beta}_j - 0}{\text{SE}[\hat{\beta}_j]}$$

shows how far away $\hat{\beta}_j$ is away from zero, normalized by its error $\text{SE}[\hat{\beta}_j]$. One can then compute the p-value corresponding to this t and test the hypothesis. Small p-value indicates **strong** relationship.

3.2 Multiple Linear Regression

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \epsilon.$$

The estimate of the coefficients $\hat{\beta}_j$, $j \in \mathbb{Z}_{p+1}$ are found by using the same least square method to minimize RSS. we interpret β_j as the *expected* (average) effect on Y with one unit increase in X_j, **holding all other predictors fixed**. This interpretation is based on the assumptions that *the predictors are uncorrelated*, so *each predictor can be estimated and tested separately.* When there are correlations among predictors, the variance of all coefficients tends to increase, sometimes dramatically, and the previous interpretation becomes hazardous because when X_j changes, everything else changes.

3.2.1 Model Assumption

- linearity: Y is linear in X. The change in Y associated with one unit of change in X_j is constant, regardless of the value of X_j. This can be examined visually by plotting the *residual plot* (e_i vs. x_i for $p = 1$ or e_i vs $\hat{y}_i$ for multiple regression). If the linear assumption is true, then the residual plot should not exhibit obvious pattern. If there is a nonlinear relationship suggested by by the residual plot, then a simple approach is to include transformed X, such as $\log X$, $\sqrt{X}$, or X^2.
- additive: The association between X_j and Y is independent of other predictors.
- Errors ϵ_i are uncorrelated. This means ϵ_i provides no information for ϵ_{i+1}. Otherwise (for example, frequently observed in a time series, where error terms are positively correlated, and *tracking* is observed in the residuals, i.e., adjacent error terms take similar values), the estimated standard error will tend to be underestimated, hence leading less confidence in the estimated model.
- Homoscedasticity: $\text{Var}(\epsilon_i) = \sigma^2$. The error terms have constant variance. If not (heteroscedasticity), one may use transformed Y, such as $\sqrt{Y}$, or $\log(Y)$ to mitigate this; or use *weighted least squares* if it's known that for example $\sigma_i^2 = \sigma^2/n_i$.

- Non-colinearity: two variables are colinear if they are highly correlated with each other. Co-linearity causes a great deal of *uncertainty* in the coefficient estimates, that is, reducing the accuracy of the coefficient estimates, thus cause the standard error of β_j to grow, and hence smaller t-statistic. As a result, we may fail to reject $H_0 : \beta_j = 0$. This in turn means the power of Hypothesis test, the probability of correctly detecting a *non-zero* coefficient is reduced by colinearity. To detect colinearity,

  - use the correlation matrix of predictors. Large value of the matrix in absolute value indicates highly correlated variable pairs. But this approach cannnot detect *multicolinearity*.
  - Use VIF (Variance inflation factor, VIF ≥ 1) to detect multicolinearity. It is possible for colinearity exists between three or more variables even if no pair of variables has a particularly high correlation. This is the *multicolinearity* situation.

  VIF is the ratio of the variance of $\hat{\beta}_j$ when fitting the full model divided by the variance of $\hat{\beta}_j$ if fit on its own. It can be calculated by

  $$\text{VIF}(\hat{\beta}_j) = \frac{1}{1 - R^2_{X_j|X_{-j}}}$$

  Where $R^2_{X_j|X_{-j}}$ is the R^2 from a regression of X_j onto all of the other predictors. A VIF value exceeds 5 or 10 (i.e., $R^2_{X_j|X_{-j}}$ close to 1) indicates colinearity.

  To remedy a colinearity problem:

  - drop a redundant variable (variables with colinearity should have similar VIF values.)
  - Combine the colinear variables into a single predictor, e.g., taking the average of the standardized versions of those variables.

Claims of causality should be avoided for observational data.

3.2.2 Assessing existence of linear relationship

- test Hypothesis (test if there is a linear relationship between the response and predictors)

  $$H_0 : \beta_1 = \beta_2 = \cdots = \beta_p = 0$$

  $$H_a : \text{at least one } \beta_j \text{ is non-zero.}$$

  using F-statistic

  $$F = \frac{\text{SSB}/\text{df(B)}}{\text{SSW}/\text{df(W)}} = \frac{(\text{TSS} - \text{RSS})/p}{\text{RSS}/(n-p-1)} \sim F_{p,n-p-1}$$

If H_0 is true, $F \approx 1$; if H_a is true, $F >> 1$. F-statistic adjust with p. Note that one **cannot conclude** if an individual t-statistic is significant, then there is at least one predictor is related to the response, especially when p is large. This is related to *multiple testing*. The reason is that when p is large, there is α (eg 5%) chance that a predictor will have a small p-value by chance. When $p > n$, F-statistic cannot be used.

If the goal is to test that a particular subset of q of the coefficients are zero, that is, (for convenience, we put the q variables chosen at the end of the variabale list)

$$H_0 : \beta_{p-q+1} = \beta_{p-q+2} = \cdots = \beta_p = 0 \tag{3.2}$$

In this case, use

$$F = \frac{(\text{RSS}_0 - \text{RSS})/q}{\text{RSS}/(n-p-1)} \sim F_{q,n-p-1}$$

where, RSS_0 is the residual sum of squares of a second model that uses all variables *except* those last q variables. When $q = 1$, F-statistic in Equation 3.2 is the square of the t-statistic of that variable. The t-statistic reported in a regression model gives the *partial effect* of adding that variable, while holding other variables fixed.

3.2.3 Assess the accuracy of the future prediciton

- confidence interval: Indicate how far away $\hat{Y} = \hat{f}(X)$ is from the population average $f(X)$ because the coefficients $\hat{\beta}_j$ are estimated, It quantifies *reducible error* around the predicted average response $\hat{f}(X)$, does-not include ϵ.
- prediction interval: Indicate how far away $\hat{Y} = \hat{f}(X)$ is from Y. predict an individual response $Y \approx \hat{f}(X) + \epsilon$. Prediction interval is always wider than the confidence interval, because it includes *irreducible error* ϵ.

3.2.4 Assessing the overall accuracy of the model

- RSE. To this end, first define the *lack of fit* measure **Residual Standard Error**

$$\text{RSE} = \sqrt{\frac{1}{n-p-1}\text{RSS}} = \sqrt{\frac{1}{n-p-1}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \approx \sigma = \sqrt{\text{Var}(\epsilon)}$$

  It is the *average amount* in $\hat{Y}$ that a response deviates from the *true regression line* $(\beta_0 + \beta_1 X)$. Note, RSE can increase with more variables if the decrease of RSS doesnot offset the increase of p.

- Approach 2: Using *R-squared* (fraction of variance in Y explained by X), which is independent of of the scale of Y, and $0 \leq R^2 \leq 1$:

$$R^2 = \frac{\text{TSS} - \text{RSS}}{\text{TSS}} = 1 - \frac{\text{RSS}}{\text{TSS}}$$

  where, $\text{TSS} = \sum_{i=1}^{n}(y_i - \bar{y})$. When R^2 is near 0 indicates that 1) either the linear model is wrong 2) or th error variance σ^2 is high, or both. R^2 measures the linear relationship between X and Y. If computed on the training set, when adding more variables, the RSS always decrease, hence R^2 will always increase.

For simple linear regression, $R^2 = r_{xy}^2$, where the *sample correlation* measures the linear relationship between variables X and Y. See the formula r_{xy} above Equation 3.1. For multiple linear regression, $R^2 = (\text{Cor}(Y, \hat{Y}))^2$. The fitted linear model maximizes this correlation among all possible linear models.

3.3 Model Selection/Variable Selections: balance training errors with model size

- **All subsets (best subsets) regression**: compute the least square fit for all 2^p possible subsets and then choose among them based on certain criterion that balance training error and model size
- **Forward selection**: Start from the *null model* that only contains β_0. Then find the best model containing one predictor that minimizing RSS. Denote the variable by β_1. Then continue to find the best model with the lowest RSS by adding one variable from the remaining predictors, and so on. Continue until some stopping rule is met: e.g., when all remaining variables have a p-value greater than some threshold.
- **Backward selection**: start with all variables in the model. Remove the variable with the largest p-value (least statistically significant). The new $(p-1)$ model is fit, and remove the variable with the largest p-value. Continue until a stopping rule is satisfied, e.g., all remaining variables have p-value less than some threshold.
- **Mixed selection**: Start with forward selection. Since the p-value for variables can become larger as new predictors are added, at any point if the p-value of a variable in the model rises above a certain threshold, then remove that variable. Continue to perform these forward and backward steps until all variables in the model have a sufficiently low p-value, and all variables outside the model would have a large p-value if added to the model.

  Backward selection cannot be used if $p > n$. Forward selection can always be used, but might include variables early that later become redundant. Mixed selection can remedy this problem.

- **others** (Chapter 6): including Mallow's C_p, AIC (Akaike Informaton Criterion), BIC, adjusted R^2, Cross-validation, test set performance.
- **not valid**: we could look at individual p-values, but when the number of variables p is large, we likely to make a false discoveries.

3.4 Handle categorical variables (factor variables)

For a categorical variable X_i with m levels, create one fewer dummy variables ($x_{ij}, 1 \leq j \leq m-1$)>. The level with no dummy variable is called the *baseline*. The coefficient corresponding to a dummy variable is the expected difference in change in Y when compared to the baseline, while holding other predictors fixed.

3.5 Adding non-linearity

3.5.1 Modeling interactions (synergy)

When two variables have interaction, then their product X_iX_j can be added into the regression model, and the product maybe considered as a single variable for inference, for example, compute its SE, t-statistics, p-value, Hypothesis test, etc.

If we include an interaction in a model, then the **Hierarchy principle** should be followed: always include the main effects, even if the p-values associated with their coefficients are not significant. This is because without the main effects, the interactions are hard to interpret, as they would also contain the main effect.

3.5.2 Adding terms of transformed predictors

1) *Polynomial regression*: Add a term involving X_i^k for some $k > 1$.
2) other forms: Adding root or logarithm terms of the predictors.

3.6 Outliers (Unusual y_i that is far from $\widehat{y}_i$)

It is typical for an outlier that does not have an unusual predictor value (with low levarage) to have little effect on the least squares fit, but it will increase RSE, hence deteriorate CI, p-value and R^2, thus affecting interpreting the model.

An outlier can be identified by computing the

$$\text{studentized residual} = \frac{e_i}{\text{RSE}_i}$$

A studentized residual great than 3 may be considered as an outlier.

3.7 High leverage points (unusual x_i)

High leverage points tend to have sizeable impact on the regression line. To quantify the observation's leverage, one needs to compute the **leverage statistic**

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{j=1}^{n}(x_j - \bar{x})^2}.$$

$1/n \leq h_i \leq 1$ and $\text{Ave}(h_i) = (p+1)/n$. A large value of this statistic (for example, great than $(p+1)/n$) indicates an observation with high leverage. The leverage $1/n \leq h_i \leq 1$, reflects the amount an observation influences its own fit.

3.8 Compared to KNN Regression

KNN regression is a non-parametric method that makes prediction at x_0 by taking the average in a K-point neightborhood

$$\hat{f}(x_0) = \frac{1}{K} \sum_{x_i \in \mathcal{N}_{x_0}} y_i$$

A small value of K provides more flexible model with low bias but high variance while a larger value of K provides smoother fit with less variance. An optimal value of K depend on the *bias-variance tradeoff*. For non-linear data set, KNN may provides better fit than a linear regression model. However, in higher dimension (e.g., $p \geq 4$), even for nonlinear data set, KNN may perform much inferior to linear regression, because of the **curse of dimensionality**, as the K observations that are nearest to x_0 may in fact far away from x_0.

3.9 Homework (* indicates optional):

- Conceptual: 1–6
- Applied: 8–15. at least one.

3.10 Code Gist

3.10.1 Python

```
dir() # provides a list of objects at the top level name space
dir(A) # display addtributes and methods for the object A
' + '.join(X.columns) # form a string by joining the list of column names by "+"
```

3.10.2 Numpy

```
np.argmax(x) # identify the location of the largest element
np.concatenate([x,y],axis=0) # concatenate two arrays x and y.
```

3.10.3 Pandas

```
X = pd.DataFrame(data=X, columns=['a','b'])

pd.DataFrame({'intercept': np.ones(Boston.shape[0]),
                  'lstat': Boston['lstat']}) # make a dataframe using a dictionary
Boston.columns.drop('medv','age') # drop the elements 'medv' and 'age' from the list of colu

pd.DataFrame({'vif':vals},
                   index=X.columns[1:]) # form a df by specifying index labels

X.values  # Convert dataframe X to numpy array
X.to_numpy() # recommended to replace the above method
DataFrame.corr(numeric_only=True) # correlations between columns
x.sort_values(ascending=False)
pd.to_numeric(auto_df['horsepower'], errors='coerce') # if error, denote it by "NaN".
auto_df.dropna(subset= ['horsepower', 'mpg',], inplace=True) # looking for NaN in the columns

auto_df.drop('name', axis=1, inplace=True)

left2.join(right2, how="left") #join two databases by index.
left1.join(right1, on="key") # left-join by left1["key"] and the index of right1.
pd.concat([s1, s4], axis="columns", join="outer")
```

3.10.4 Graphics

```
xlim = ax.get_xlim() # get the x_limit values xlim[0], xlim[1]
ax.axline() # add a line to a plot
ax.axhline(0, c='k', ls='--'); # horizontal line
line, = ax.plot(x,y,label="line 1") # "line 1" is the legend
# alternatively the label can be set by
line.set_label("line 1")
ax.scatter(fitted, residuals, edgecolors = 'k', facecolors = 'none')
ax.plot([min(fitted),max(fitted)],[0,0],color = 'k',linestyle = ':', alpha = .3)
ax.legend(loc="upper left", fontsize=25) # adding legendes
ax.annotate(i,xy=(fitted[i],residuals[i])) # annote at the xy position with i.


plt.style.use('seaborn') # pretty matplotlib plots
plt.rcParams.update({'font.size': 16})
plt.rcParams["figure.figsize"] = (8,7)

plt.rc('font', size=10)
plt.rc('figure', titlesize=13)
plt.rc('axes', labelsize=10)
plt.rc('axes', titlesize=13)
plt.rc('legend', fontsize=8) # adjust legend globally
```

3.10.5 Using Sns

```
sns.set(font_scale=1.25) # set font size 25% larger than default
sns.heatmap(corr, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10})
ax = sns.regplot(x=x, y=y)
```

3.10.6 Using Sklearn

```
from sklearn.linear_model import LinearRegression
## Set the target and predictors
X = auto_df['horsepower']

### To get polynomial features
poly = PolynomialFeatures(interaction_only=True,include_bias = False)
X = poly.fit_transform(X)
```

```
y = auto_df['mpg']

## Reshape the columns in the required dimensions for sklearn
length = X.values.shape[0]
X = X.values.reshape(length, 1) #both X and y needs to be 2-D
y = y.values.reshape(length, 1)

## Initiate the linear regressor and fit it to data using sklearn
regr = LinearRegression()
regr.fit(X, y)
regr.intercept_
regr.coef_

pred_y = regr.predict(X)
```

3.10.7 Using statsmodels and ISLP

```
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize,
                         poly)

import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.stats.outliers_influence \
     import variance_inflation_factor as VIF
from statsmodels.stats.anova import anova_lm

#Training
Boston = load_data("Boston")
#hand-craft the design matrix X
X = pd.DataFrame({'intercept': np.ones(Boston.shape[0]), #design matrix. intercept column
                  'lstat': Boston['lstat']})
#the following is the preferred method to create X
design = MS(['lstat']) # specifying the model variables. Automatically add an intercept, addi
design = design.fit(Boston) # do intial computation as specified in the model object design l

X = design.transform(Boston) # apply the fitted transformation to the data to create X
#alternatiely,
X = design.fit_transform(Boston) # this combines the .fit() and .transform() two lines

y = Boston['medv']
```

```
model = sm.OLS(y, X) # setup the model
model = smf.ols('mpg ~ horsepower', data=auto_df) # alternatively use smf formula, y~x
smf.ols("y ~ x -1" , data=df).fit() # "-1" not inclding the intercept
results = model.fit() # results is a dictionary:.summary(), .params

results.summary()
results.params # coefficients
results.resid # reisdual array
results.rsquared # R^2
results.pvalues
np.sqrt(results.scale) # RSE
results.fittedvalues # fitted \hat(y)_i at x_i in the traning set


summarize(results) # summzrize() is from ISLP to show the esstial results from model.fit()

# Makding prediciton
new_df = pd.DataFrame({'lstat':[5, 10, 15]})  # new test-set containing data where to make pr
newX = design.transform(new_df) # apply the same transform to the test-set
new_predictions = results.get_prediction(newX);
new_predictions.predicted_mean #predicted values
new_predictions.conf_int(alpha=0.05) #for the predicted values

new_predictions.conf_int(obs=True, alpha=0.05) # prediction intervals by setting obs=True

# Including an interaction term
X = MS(['lstat',
        'age',
        ('lstat', 'age')]).fit_transform(Boston) #interaction term ('lstat', 'age')

# Adding a polynomial term of higher degree
X = MS([poly('lstat', degree=2), 'age']).fit_transform(Boston) # Note poly is from ISLP, # ac
# Given a qualitative variable, `ModelSpec()` generates dummy
variables automatically, to avoid collinearity with an intercept, the first column is droppe

# Compare nested models using ANOVA
anova_lm(results1, results3) # result1 is the result of linear model, an result3 is the resul

# Identify high leverage x
infl = results.get_influence()
# hat_matrix_diag calculate the leverate statistics
np.argmax(infl.hat_matrix_diag) # identify the location of the largest levarage
```

```
# Calculate VIF
vals = [VIF(X, i)
        for i in range(1, X.shape[1])] #excluding column 0 because it's all 1's in X.
vif = pd.DataFrame({'vif':vals},
                   index=X.columns[1:])
vif # VIF exceeds 5 or 10 indicates a problematic amount of colinearity
```

Useful Code Snippets

```
def abline(ax, b, m, *args, **kwargs):
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
    ylim = [m * xlim[0] + b, m * xlim[1] + b]
    ax.plot(xlim, ylim, *args, **kwargs)

# Plot scatter plot with a regression line
ax = Boston.plot.scatter('lstat', 'medv')
abline(ax,
       results.params[0],
       results.params[1],
       'r--',
       linewidth=3)

# Plot residuals vs. fitted values (note, not vs x, therefore works for multiple regression)
ax = subplots(figsize=(8,8))[1]
ax.scatter(results.fittedvalues, results.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--');

# Alternatively
sns.residplot(x=X, y=y, lowess=True, color="g", ax=ax)

# Plot the smoothed residuals~fitted by LOWESS
from statsmodels.nonparametric.smoothers_lowess import lowess
smoothed = lowess(residuals,fitted) # Note the order (y,x)
ax.plot(smoothed[:,0],smoothed[:,1],color = 'r')

# QQ plot for the residuas (obtain studentized residuals for identifying outliers)
import scipy.stats as stats
sorted_student_residuals = pd.Series(smf_model.get_influence().resid_studentized_internal)
```

```
sorted_student_residuals.index = smf_model.resid.index
sorted_student_residuals = sorted_student_residuals.sort_values(ascending = True)
df = pd.DataFrame(sorted_student_residuals)
df.columns = ['sorted_student_residuals']

#stats.probplot() #assess whether a dataset follows a specified distribution
df['theoretical_quantiles'] = stats.probplot(df['sorted_student_residuals'], dist = 'norm', 

x = df['theoretical_quantiles']
y = df['sorted_student_residuals']
ax.scatter(x,y, edgecolor = 'k',facecolor = 'none')


# Plot leverage statistics
infl = results.get_influence()
ax = subplots(figsize=(8,8))[1]
ax.scatter(np.arange(X.shape[0]), infl.hat_matrix_diag)
ax.set_xlabel('Index')
ax.set_ylabel('Leverage')
np.argmax(infl.hat_matrix_diag) # identify the location of the largest levarage
```

4 Chapter 4: Classification

Given a feature vector X and a *qualitative* (categorical) response Y taking finite values in a set $\mathcal{C}$, the classification task is to build a classifier $C(X)$ that takes an input X and predicts its class $Y = C(X) \in \mathcal{C}$. This is often done by model $P(Y = k|X = x)$ for each $k \in \mathcal{C}$.

4.1 Linear regression and Classification

- For a *binary* classification, one can use linear regression and does a good job. In this case, the linear regression classifier is equivalent to LDA, because

  $$P(Y = 1|X = x) = E[Y|X = x]$$

  However, linear regression may not represent a probability as it may give a value outside the interval $[0, 1]$.
- When there are more than two classes, linear regression is not appropriate, because any chosen coding of the Y variable imposes an ordering and fixed differences among categories, which may not be implied by the data set. If the coding changes, a dramatic function will be fitted, which is not reasonable. One should turn to *multiclass logistic regression* or *Discriminant Analysis*.

4.2 Logistic Regression

Logistic regression is a *discriminative learning*, because it directly calculates the conditional probability $P(Y|X)$ to make classification.

4.2.1 Binary classification

with a single variable Logistic regression simply convert the linear regression to probability by

$$p(X) = Pr(Y = 1|X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}.$$

Note the *logit* or *log odds* is linear

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X.$$

Increasing X by one unit, changes the log odds by β_1. Equivalently, it multiplied the odds by e^{β_1}. The rate of change of $p(X)$ is no longer a constant, but depends on the current value of X. Positive β_1 implies increasing $p(X)$, and vice vesa.

The parameters are estimated by maximizing the *liklihood*

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i:y_i=0} (1 - p(x_i))$$

With the estimated parameters $\hat{\beta}_j, j = 0, 1$, one can calculate the probability

$$p(X) = Pr(Y = 1|X) = \frac{e^{\hat{\beta}_0+\hat{\beta}_1 X}}{1 + e^{\hat{\beta}_0+\hat{\beta}_1 X}}$$

4.2.2 with multiple variables

In this case, simply let the logit be a linear function of p variables.

Note when there are multiple variables, it's possible to have variables confounding (especially when two variables are correlated): the coefficient of a variable may changes significantly or may change sign, this is because the coefficient represents the rate of change in Y of that variable when holding other variable constants. The coefficient reflects the effect when other variables are hold constant, how the variable affects Y, and this effect may be different than when only this variable is used in the model.

> **Note**
>
> One can include a nonlinear term such as a quadratic term in the logit model, similar to a linear regression that includes a non-linear term.

4.2.3 Multi-class logistic regression (multinomial regression) with more than two classes

in this case, we use the *softmax* function to model

$$\Pr(Y = k|X) = \frac{e^{\beta_{0k}+\beta_{1k}X_1+\cdots+\beta_{pk}X_p}}{\sum_{\ell=1}^{K} e^{\beta_{0\ell}+\beta_{1\ell}X_1+\cdots+\beta_{p\ell}X_p}} = a_k$$

for each class k. Note $\Sigma_k a_k = 1$ and the cross-entropy loss function is given by $-\log \ell(\beta) = -\Sigma_k \mathbb{1}_k \log a_k$, where β represents all the parameters.

The log odds between kth and k'th classes equals

$$\log(\frac{\Pr(Y=k|X=x)}{\Pr(Y=k'|X=x)}) = (\beta_{k0} - \beta_{k'0}) + (\beta_{k1} - \beta_{k'1}) + \cdots + (\beta_{kp} - \beta_{k'p})$$

4.3 Discriminant Classifier: Approximating Optimal Bayes Classifier

Apply the Bayes Theorem, the model

$$\Pr(Y=k|X=x) = \frac{\Pr(X=x|Y=k)\cdot\Pr(Y=k)}{\Pr(X=x)} = \frac{\pi_k f_k(x)}{\sum_{\ell=}^{K} \pi_\ell f_\ell(x)}$$

where π_k = Pr(Y=k) is the *marginal* or *prior* probability for class k, and $f_k(x) = \Pr(X = x|Y=k)$ is the *density* for X in class k. Note the denominator is a *normalizing constant*. So when making decisions, effectively we compare $\pi_k f_k(x)$, and assign x to a class k with the largest $\pi_k f_k(x)$.

Discriminant uses the full liklihood $P(X,Y)$ to calculate $P(Y|X)$ to make a classification, so it's known as *generative learning*.

- when f_k is chosen as a normal distribution with constant variance (σ^2) for $p = 1$ or correlation matrix Σ for $p > 1$, this leads to the LDA. For $p = 1$, the *discriminant score* is given by

  $$\delta_k(x) = x \cdot \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + \log(\pi_k)$$

  when $K = 2$ and $\pi_1 = \pi_2 = 0.5$m then the *decision boundary* is given by

  $$x = \frac{\mu_1 + \mu_2}{2}.$$

  When $p \geq 2$, assume that $X = (X_1, X_2, \cdots, X_p)$ is drawn from a multivariate Gaussian distribution $X \sim N(\mu_k, \Sigma)$, with a class-specific mean vector and a a common variance matrix.

  $$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log\pi_k = c_{k0} + c_{k1}x_1 + \cdots + c_{kp}x_p.$$

  The score function (posterior probability) is *linear* in x. With $\hat{\delta}_k(x)$ for each k, it can be converted to the class probability by the *softmax* function

  $$\hat{\Pr}(Y=k|X=x) = \frac{e^{\hat{\delta}_k(x)}}{\sum_{\ell=1}^{K} e^{\hat{\delta}_\ell(x)}}$$

The π_k, μ_k and σ are estimate the follwing way:

$$\hat{\pi}_k = \frac{n_k}{n}$$

$$\hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i$$

$$\hat{\sigma}^2 = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2 = \sum_{k=1}^{K} \frac{n_k - 1}{n-K} \hat{\sigma}_k^2$$

where $\hat{\sigma}_k^2 = \frac{1}{n_k - 1} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2$ is the estimated variance for the k-th class.

> **Note**
>
> One can include a nonlinear term such as a quadratic term in the LDA model, similar to a linear regression that includes a non-linear term.

- when each class chooses a different Σ_k, then it's QDA. It assumes an observation from the k-th class is $X \sim N(\mu_k, \Sigma_k)$.The score function has a *quadratic* term

  $$\delta_k(x) = -\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k) + \log \pi_k - \frac{1}{2} \log |\Sigma_k|$$

  QDA has much more parameters $Kp(p+1)/2$ to estimate compared to LDA (Kp), hence has higher flexibility and may lead to higher variance. When there are few training examples, LDA tend to perform better and reducing variance is crucial. When there is a large traning set, QDA is recommended as variance is not a major concern. LDA is a special case of QDA.

- when the features are modeled independently, i.e., there is no association between the p predictors, $f_k(x) = \prod_{j=1}^{p} f_{jk}(x_j)$, the method is *naive Bayes*, and Σ_k are diagonal. Any classifier with a linear decision boundary is a special case of NB. So LDA is a special case of NB. To estimate f_kj, one can

  - assume that $X_j|Y = k \sim N(\mu_{jk,\sigma_{jk}^2})$, that is, a class specific covariance but is diagonal. QDA's Σ_k is not diagonal. If we model $f_{kj}(x_j) \sim N(\mu_{kj} + \sigma_j^2)$ (Note σ_j^2 is shared among clases), In this case NB is a special case of LDA that has a diagonal Σ and
  - use a non-parametric estimate such as histogram (or a smooth kernel density estimator) for the observations of the jth Predictor within each class.
  - If X_j is qualitative, then one can simply count the proportion of training observations for the jth predictor corresponding to each class.

- Can applied to *mixed* feature vectors (qualitative and quantitative). NB does not assume normally distributed predictors.
- Despite strong assumptions, performs well, especially when n is not large enough relative to p, when estimating the joint distribution is difficult. It introduces some biases but reduces variance, leading to a classifier that works quite well as a result of bias-variance trade-off.
- Useful when p is very large.
- NB is a *generalized additive model*.
- Neigher NB nor QDA is a special case of the other. Because QDA contains interaction term $x_i x_j$, while NB is purely additive, in the sense that a function of x_i is added to a function of x_j. Therefore, QDA potentially is a better fit when the interactions among predictors are important.

4.3.1 Why discriminant analysis

- When the classes are well-separated, the parameter estimation of logistic regression is unstable, while LDA does not suffer from this problem.
- if the data size n is small and the distribution of X is approximately normal in each of the classes, then LDA is more stable than logistic regression. Also used when $K > 2$.
- when there are more than two classes, LDA provides low-dimensional views of the data hence popular. Specifically, when there are K classes, LDA can be viewed exactly in $K-1$ dimensional plot. This is because it essentially classifies to the closest centroid, and they span a $K-1$ dimensional plane.
- For a two-class problem, the logit of $p(Y=1|X=x)$ by LDA (generative learning) is a linear function in X, the same as a logistic regression (discriminative learning). The difference lies in how the parameters are estimated. But in practice, they are similar.
- LDA assumes the predictors follow a multivariable normal distribution with a shared Σ among classes. So when this assumption holds, we expect LDA performs better; and Logistic regress performs better when this asuumption does not hold.

4.4 KNN

KNN is a non-parametric method and doesnot assume a shape for the decision boundary. KNN assign the class of popularity to $X = x$ in a K-neighborhood.

- KNN dominates LDA and Logistic Regression when the decision boundary is highly non-linear, provided n is large and p is small. As KNN breaks down when p is large.

- KNN requires large $n >> p$, this is because KNN is non-parametric and tends to reduce bias but increase variance.
- When the decision boundary is non-linear but n is only modest and p is not very small, QDA may outperform KNN. This is because QDA provides a non-linear boundary while taking advantage of a parametric form, which means that if requires smaller size for accurate classification.
- Unlike logistic regression, KNN does not tell which predictors are more importnat: We dont get a table of coefficients.
- When the decision boundary is linear, LDA or logistic regression may perform better, when the boundary is moderately non-linear, QDA or NB may perform better; For a much more complicated decision boundary, KNN may perform better.

4.5 Poisson Regression

When Y is discrete and non-negative, a linear regression model is not satisfactory, even with the transformation of $\log(Y)$, because log does not allow $Y = 0$.

- Poisson Regression: typically used to model counts,

$$\Pr(Y = k) = \frac{e^{-\lambda}\lambda^k}{k!}, \qquad k = 0, 1, 2, \cdots,$$

  where, $\lambda = E(Y) = \text{Var}(Y)$. This means that if Y follows a Poissson distribution, the larger the mean of Y, the larger its variance. Posisson regression can handle this when variance changes with mean, but linear regression cannot, because it assumes constant variance.

Assume

$$\log(\lambda(X_1, X_2, \cdots, X_p)) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

Then one can use maximum likelihood

$$\ell(\beta_0, \beta_1, \cdots, \beta_p) = \prod_{i=1}^{n} \frac{e^{-\lambda(x_i)}\lambda(x_i)^{y_i}}{y_i!}$$

to estimate the parameters.

- Interpretation: An increase in X_j by one unit is associated with a change in $E(Y) = \lambda$ by a *factor* (percentage) of $\exp(\beta_j)$.

4.6 Generalized Linear Models (GLM)

Perform a regression by modeling Y from a particular member of the *exponential family* (Gaussian, Bernoulli, Poisson, Gamma, negative binomial), and then transform the mean of Y to a linear function.

- Use predictors $X_1, \cdots, X_p$ to predict Y. Assume Y conditional on X follow some distribution: For linear regression, assume Y follows a normal distribution; for logistic regression, assume Y follows a Bernoulli (multinomial distribution for multi-class logistic regression) distribution; For poisson distribution, assume Y follows a poisson distribution.
- Each approach models the mean of Y as a function of X using a *linking function* η to transform $E[Y|X]$ to a linear function.
  - for linear regression

    $$E(Y|X) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

    $\eta(\mu) = \mu$
  - for logistic regression

    $$E(Y|X) = P(Y = 1|X) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}$$

    $\eta(\mu) = \log(\mu/(1-\mu))$
  - for Poisson regression

    $$E(Y|X) = \lambda(X) = e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}$$

    $\eta(\mu) = \log(\mu)$.
  - Gamma regression and negative binomial regression.

4.7 Assessment of a classifier

- Confusion matrix
- Overall error rate: equals to

  $$\frac{FP + FN}{N + P}$$

- Class-specific performance: One can adjust the decision boundary (posterior probability threshold) to improve class specific performance at the expense of lowered overall performance.

- percentage of TP detected among all positives

  $$\text{sensitivity (recall, power)} = TPR = \frac{TP}{TP+FN} = \frac{TP}{P} = 1 - \text{Type II error} = 1-\beta$$

  this is equal to $1-FNR$, where, FNR is The fraction of positive examples that are classified as negatives

  $$FNR = \frac{FN}{FN+TP} = \frac{FN}{P}$$

- percentage of TN detected among all negatives

  $$\text{specificity} = TNR = \frac{TN}{TN+FP} = \frac{TN}{N}$$

  This is equal to $1-FPR$, where, False positive rate (FPR): the fraction of negative examples (N) that are classified as positive:

  $$FPR = \frac{FP}{FP+TN} = \frac{FP}{N} = \text{Type I error}(\alpha)$$

- ROC (receiver operating characteristic curve): plot true positive rate (TPR=1-Type II error) ~ false positive rate (FPR= 1- specificity=Type I error) as a threshold for the posterior probability of positive class changes from 0 to 1. The point on the ROC curve closest to the point (0,1) corresponds to the best classifier.

- AUC (area under the ROC): Overall performance of a classifier summarized over all thresholds. AUC measures the probability a random positive example is ranked higher than a random negative example. A larger AUC indicates a better classifier. More specifically, define the score function for th e i-th observation by $Pr(Y = 1|X = x_i)$. Consider all pairs consisting of one observation in Class 1 and one observation in Class 0, then the AUC is the fraction (probability) of pairs for which the score for the observation in Class 1 exceeds the score for the observations in Class 0.

- class-specific prediction performance

  -

  $$\text{precision} = \frac{TP}{TP+FP} = \frac{TP}{\text{predicted postives}} = 1 - \text{false discovery proportion}$$

4.8 Homework:

- Conceptual: 1,2,3,4, 5,6,7,8, 9, 10, 12
- Applied: 13, 14*,15*,16*

4.9 Code Gist

4.9.1 Python

4.9.2 Numpy

```
np.where(lda_prob[:,1] >= 0.5, 'Up','Down')
np.argmax(lda_prob, 1) #argmax along axis=1 (col)
np.asarray(feature_std) # convert to np array
np.allclose(M_lm.fittedvalues, M2_lm.fittedvalues)
#check if corresponding elts are equal within rtol=1e-5 and atol=-1e08
```

4.9.3 Pandas

```
Smarket.corr(numeric_only=True)
train = (Smarket.Year < 2005)
Smarket_train = Smarket.loc[train] # equivalent to Smarket[train]
Purchase.value_counts() # frequency table
feature_std.std() #calculate column std
S2.index.str.contains('mnth')
Bike['mnth'].dtype.categories # get the categories of the categorical data
obj2 = obj.reindex(["a", "b", "c", "d", "e"])# rearrange the entries in obj according to the
```

4.9.4 Graphics

```
ax_month.set_xticks(x_month) # set_xticks at the place given by x_month
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20)
ax.axline([0,0], c='black', linewidth=3,
          linestyle='--', slope=1);#axline method draw a line passing a given point with a g
```

4.9.5 ISLP and Statsmodels

```
from ISLP import confusion_table
from ISLP.models import contrast

# Logistic Regression using sm.GLM() syntax similar to sm.OLS()
design = MS(allvars)
X = design.fit_transform(Smarket)
y = Smarket.Direction == 'Up'
glm = sm.GLM(y,
```

```
            X,
            family=sm.families.Binomial())
results = glm.fit()
summarize(results)
results.pvalues
probs = results.predict() #without data set, calculate predictions on the training set.
results.predict(exog=X_test) # on test set
# Prediction on a new dataset
newdata = pd.DataFrame({'Lag1':[1.2, 1.5],
                        'Lag2':[1.1, -0.8]});
newX = model.transform(newdata)
results.predict(newX)
confusion_table(labels, Smarket.Direction) #(predicted_labels, true_labels)
np.mean(labels == Smarket.Direction) # calculate the accuracy

hr_encode = contrast('hr', 'sum') #coding scheme for categorical data: the unreported coeffi

#Poisson Regression
M_pois = sm.GLM(Y, X2, family=sm.families.Poisson()).fit()
#`family=sm.families.Gamma()` fits a Gamma regression
model.
```

4.9.6 sklearn

```
from sklearn.discriminant_analysis import \
     (LinearDiscriminantAnalysis as LDA,
      QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

#LDA
lda = LDA(store_covariance=True) #store the covariance of each class
X_train, X_test = [M.drop(columns=['intercept']) # drop the intercept column
                   for M in [X_train, X_test]]
lda.fit(X_train, L_train) # LDA() model will automatically add a intercept term

lda.means_ # mu_k (n_classes, n_features)
lda.classes_
lda.priors_ # prior probability of each class
```

```
#Linear discrimnant vectors
lda.scalings_ #Scaling of the features in the space spanned by the class centroids. Only ava

lda_pred = lda.predict(X_test) #predict class labels
lda_prob = lda.predict_proba(X_test) #ndarray of shape (n_samples, n_classes)

#QDA
qda = QDA(store_covariance=True)
qda.fit(X_train, L_train)
qda.covariance_[0] #estimated covariance for the first class

# Naive Bayes
NB = GaussianNB()
NB.fit(X_train, L_train)
NB.class_prior_
NB.theta_ #means for (#classes, #features)
NB.var_ #variances (#classes, #features)
NB.predict_proba(X_test)[:5]

# KNN
knn1 = KNeighborsClassifier(n_neighbors=1)
X_train, X_test = [np.asarray(X) for X in [X_train, X_test]]
knn1.fit(X_train, L_train)
knn1_pred = knn1.predict(X_test)

# When using KNN one should standarize each varaibles
scaler = StandardScaler(with_mean=True,
                        with_std=True,
                        copy=True) # do calculaton on a copy of the dataset
scaler.fit(feature_df)

#train test split
X_std = scaler.transform(feature_df)
(X_train,
 X_test,
 y_train,
 y_test) = train_test_split(np.asarray(feature_std),
                            Purchase,
                            test_size=1000,
                            random_state=0)

# Logistic Regression
logit = LogisticRegression(C=1e10, solver='liblinear') #use solver='liblinear'to avoid warni
```

```
logit.fit(X_train, y_train)
logit_pred = logit.predict_proba(X_test)
```

4.9.7 Useful code snippet

```
# Tuning KNN
for K in range(1,6):
    knn = KNeighborsClassifier(n_neighbors=K)
    knn_pred = knn.fit(X_train, y_train).predict(X_test)
    C = confusion_table(knn_pred, y_test)
    templ = ('K={0:d}: # predicted to rent: {1:>2},' +  # > for right alighment
            '  # who did rent {2:d}, accuracy {3:.1%}')
    pred = C.loc['Yes'].sum()
    did_rent = C.loc['Yes','Yes']
    print(templ.format(
          K,
          pred,
          did_rent,
          did_rent / pred))
```

5 Chapter 5: Resampling Methods

Resampling methods are mainly used to estimate the test error by resampling the training set. Two methods: cross-validation and bootstrap. Theses methods refit a model to samples from the training set, in order to obtain additional information (eg. prediction error on the test set, standard deviation and bias of estimated parameters) about the fitted model.

Recall training error in general *dramatically underestimate* the test error, and in general, training error decreases as the model flexibility increases, but the test error shows a characteristic U-curve due to the bias-variance trade-off of the test error.

Model Assessment: evaluating a model's performance *Model selection*: selecting the proper level of flexibility.

5.1 how to estimate test error

- use a large designated test set, but often not available.
- make adjustment to the training error to estimate the test error, e.g., Cp statistic, AIC and BIC.
- validation set approach: estimate the test error by *holding out* a subset of the training set, also called a *validation set*.
  - the estimate of the test error can be highly variable, depending on the random train-validation split.
  - Only a subset of the training set is used to fit the model. Since statistical methods tend to perform worse when trained on a smaler data set, which suggests the validation error tends to *overestimate* the test error compared to the model that uses the entire training set.
- K-fold Cross-Validation: randomly divide the data into K equal-sized parts $C_1, C_2, \cdots, C_K$. For each k, leave out part k, fit the model on the remaining $K-1$ parts (combined, $(K-1)/K$ of the original traning set), and then evaluate the model on the part k. Then repeat this for each k, and weighted average of the errors is computed:

$$CV_{(K)} = \sum_{k=1}^{K} \frac{n_k}{n} \text{MSE}_k$$

where $\text{MSE}_k = \sum_{i \in C_k} (y_i \neq \hat{y}_i)/n_k$.

For classification problem, simply replace MSE_k with the misclassificaiton rate $\text{Err}_k = \sum_{i \in C_k} I(y_i \neq \hat{y}_i)/n_k$.

The estimated standard error of CV_k can be calculated by

$$\hat{\text{SE}}(CV_k) = \sqrt{\frac{1}{K}\sum_{k=1}^{K}\frac{(\text{Err}_k - \overline{\text{Err}_k})^2}{K-1}}$$

The estimated error tends bias upward because it uses only $(K-1)/K$ of the training set. This *bias is minimized with $K = n$ (LOOCV)*, but LOOCV estimate has *high variance* due to the high correlation between folds.

- LOOCV: it's a special case of K-fold CV with $K = n$. For least squares linear or polynomial regression, the LOOCV error can be computed by

  $$\text{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1 - h_i}\right)^2$$

  Where h_i is the leverage statistic of x_i. There is no randomness in the error. The leverage $1/n \leq h_i \leq 1$, reflects the amount an observation influences its own fit. The above formula doesn't hold in genearl, in which case the model has to refit n times to estimate the test error.

- for LOOCV, the estimate from each fold are highly correlated, hence their average can have high variance.

- better choice is $K = 5$ or $K = 10$ for bias-variance trade-off, because large k leads to low bias but high variance due to the increased correlation between models. Despite the estimated test error sometimes *underestimate* the true test error, they then to be close to identify the correct flexibility where the test error is minimum.

- Bootstrap: Primarily used to estimate the standard error, or a CI (called *bootstrap percentile*) of an estimate . Repeatedly sampling the training set with replacement and obtain a *bootstrap set* of the *the same size* as the original training set. One can fit a model and estimate a parameter with each bootstrap data set, and then estimate the *standard error* using the estimated parameters by the bootstrap model, assuming there are B bootstrap data sets:

  $$SE_B(\hat{\alpha}) = \sqrt{\frac{1}{B-1}\sum_{r=1}^{B}(\hat{\alpha}^{*r} - \bar{\hat{\alpha}}^*)^2}$$

  - Note sometimes sampling with replacement must take caution, for example, one can't simply sample a time series with replacement because the data are sequential.

- Estimate prediction error: Each bootstrap sample has significant overlap with the original data, in fact, about 2/3 of the original data points appear in each bootstrap sample. If we use the original data set as the validation set, This will cause the bootstrap to seriously *underestimate* the true prediction error. To fix this, one can only use predictions on those samples that do not occur (by chance) in a bootstrap sample.
- Bootstrap vs. Permutation test: permutation methods sample from an estimated *null* distribution for the data, and use this to estimate p-values and *False Discovery Rates* for hypothesis tests.

  The bootstrap can be used to test a null hypothesis in simple situation. Eg. If $H_0 : \theta = 0$, we can check whether the confidence interval for θ contains zero.

5.2 Homework

- Conceptual: 1,2,3,4
- Applied: 5–9, at least one.

5.3 Code Gist

5.3.1 Python

```
np.empty(1000) #create an array without initializing
quartiles = np.percentile(arr, [25, 50, 75])
```

5.3.2 Numpy

```
c = np.power.outer(row, col) # mesh of row[i]^col[j] power.
# random choice
rng = np.random.default_rng(0)
alpha_func(Portfolio,
           rng.choice(100, # random numbers are selected from arange(100)
                      100, #size
                      replace=True))
```

5.3.3 Pandas

```
np.cov(D[['X','Y']].loc[idx], rowvar=False) #cov compute corr of variables. rowvar-False: co
```

5.3.4 Graphics

5.3.5 ISLP and statsmodels

```
# function that evalues MSE for training a model
def evalMSE(terms,    #predictor variables
            response, #response variable
            train,
            test):

   mm = MS(terms)
   X_train = mm.fit_transform(train)
   y_train = train[response]

   X_test = mm.transform(test)
   y_test = test[response]

   results = sm.OLS(y_train, X_train).fit()
   test_pred = results.predict(X_test)

   return np.mean((y_test - test_pred)**2)

# Compare polynomial models of different degrees
MSE = np.zeros(3)
for idx, degree in enumerate(range(1, 4)):
    MSE[idx] = evalMSE([poly('horsepower', degree)],
                       'mpg',
                       Auto_train,
                       Auto_valid)
MSE

# Estimating the accuracy of a LR model using bootstrap

# Compute the SE of the boostraped values computed by func
def boot_SE(func,
            D,
            n=None,
            B=1000,
```

```
            seed=0):
    rng = np.random.default_rng(seed)
    first_, second_ = 0, 0
    n = n or D.shape[0]
    for _ in range(B):
        idx = rng.choice(D.index,
                         n,
                         replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2
    return np.sqrt(second_ / B - (first_ / B)**2) #compute var.
def boot_OLS(model_matrix, response, D, idx):
    D_ = D.loc[idx]
    Y_ = D_[response]
    X_ = clone(model_matrix).fit_transform(D_) #clone create a deep copy.
    return sm.OLS(Y_, X_).fit().params

quad_model = MS([poly('horsepower', 2, raw=True)]) #raw=True: not normalize the feature
quad_func = partial(boot_OLS,
                    quad_model,
                    'mpg')
boot_SE(quad_func, Auto, B=1000)
```

5.3.6 sklearn

```
from functools import partial
from sklearn.model_selection import \
     (cross_validate,
      KFold,
      ShuffleSplit)
from sklearn.base import clone
from ISLP.models import sklearn_sm #wrapper to feed a sm model to sklearn

#Cross Validation
hp_model = sklearn_sm(sm.OLS,
                      MS(['horsepower']))
X, Y = Auto.drop(columns=['mpg']), Auto['mpg']
cv_results = cross_validate(hp_model,
                            X,
                            Y,
                            cv=Auto.shape[0]) #cv=K.loocv. Can use cv=KFold()object
```

```
cv_err = np.mean(cv_results['test_score']) # test_score: MSE
cv_err

# Use KFold to partition instead of using an integer.
cv_error = np.zeros(5)
cv = KFold(n_splits=10,
           shuffle=True,#shuffle before splitting
           random_state=0) # use same splits for each degree
for i, d in enumerate(range(1,6)):
    X = np.power.outer(H, np.arange(d+1))
    M_CV = cross_validate(M,
                          X,
                          Y,
                          cv=cv)
    cv_error[i] = np.mean(M_CV['test_score'])
cv_error

# using ShuffleSplit() method
validation = ShuffleSplit(n_splits=10,
                          test_size=196,
                          random_state=0)
results = cross_validate(hp_model,
                         Auto.drop(['mpg'], axis=1),
                         Auto['mpg'],
                         cv=validation)
results['test_score'].mean(), results['test_score'].std()


#View skleanrn fitting results using model.results_
hp_model.fit(Auto, Auto['mpg']) # hp_model is a sklearn model sk_model.fit(X, Y) for trainni
model_se = summarize(hp_model.results_)['std err'] #summarize is an ISLP function
model_se
```

5.3.7 Useful code snippet

6 Chapter 6: Linear Model Selection and Regrularization

Linear models are *interpretable* and often shows small variance. They are fitted by *OLS*. There are other methods that can either provide alternatives or improve linear regression models in terms of *prediction accuracy* (especially when $p > n$) and automatic *feature selection* for improved interpretability.

There are three classes of methods:

- subset selection: pick a subset of the p predictors that best explains the response.
- Shrinkage (regularization). With an added regularizing term, estimated parameters are shrunken to zero relative the OLS estimates. If L^2 regularization is used, all coefficients are shrunk toward zero by the same proportion; while if L^1 is used, then some coefficients will become zero, leading to actual *variable selection* or *sparse representation*.
- Dimension reduction. Project p-predictors to a M ($M < p$) dimensional subspace. Each new direction is a linear combination (or projection) of the p-variables. These M-projections can then be used to fit a linear regression model(**PCR** if the M-projections are obtained in an unsupervised way; or **PLR** if these M-projections are obtained in a supervised way))

6.1 Best Subset Selecttion

Algorithm

1. Fit the data with the *null model* $\mathcal{M}_0$, which contains no predictors. This model simply set $Y = \text{mean}(y_i)$.
2. for $k = 1, 2, \cdots, p$: fit $\binom{p}{k}$ models containing exactly k predictors. Pick the best one that having the smallest RSS or largest R^2 (or *deviance* for classification problem, i.e., $-2\max\log(\text{likelihood})$ on the *training set*, called $\mathcal{M}_k$. Note for each categorical variable with L-level, there are $L-1$ dummy variables.
3. Select the best one among $\mathcal{M}_0, \cdots, \mathcal{M}_p$ using cross-validation or other measures such as $C_p(AIC)$, BIC or adjusted R^2. If cross-validation is used, then Step 2 is repeated on each training fold, and the validation errors are averaged to select the best k. The the model $\mathcal{M}_k$ fit on the full training set is delivered for chosen k.

Best subset selection suffers - high computation: needs to compute 2^p models - overfitting due to the large search space of models

6.2 Stepwise selection

Both Forward and Backward selection are stepwise selection. They are used when p is large. They searches over $1 + p(p+1)/2$ models and are *greedy* algorithm and is not guaranteed to find the best possible model out of all 2^p models.

- Backward selection requires $n > p$ (so that the full model can be fit);
- Forward selection can be used when $n < p$ but only fits up to models with n variables.
- One can combine forward and backward selection to a hybrid selection.

6.2.1 Forward Stepwise Selection

Adding one variable at at time that offers the greatest additional improvement.

Algorithm

1. Fit the data with the *null model* $\mathcal{M}_0$, which contains no predictors. This model simply set $Y = \text{mean}(y_i)$.
2. for $k = 1, 2, \cdots, p-1$:

- fit all $p - k$ models that augment the predictors in $\mathcal{M}_k$ with one additional predictor.
- Pick the best one that having the smallest RSS or largest R^2 on the training set, called $\mathcal{M}_{k+1}$.

3. Select the best one among $\mathcal{M}_0, \cdots, \mathcal{M}_p$ using cross-validation or other measures such as $C_p(AIC)$, BIC or adjusted R^2.

6.2.2 Backward Stepwise Selection

It begins with the full model with all variables, and iteratively removing one variable at at time.

Algorithm

1. Fit the data with the *full model* $\mathcal{M}_p$, which contains all predictors.
2. for $k = p, p-1, \cdots, 1$:

- fit all k models that contains all but one of the predictors in $\mathcal{M}_k$.
- Pick the best one that having the smallest RSS or largest R^2 on the training set, called $\mathcal{M}_{k-1}$.

3. Select the best one among $\mathcal{M}_0, \cdots, \mathcal{M}_p$ using cross-validation or other measures such as $C_p(AIC)$, BIC or adjusted R^2.

6.3 Model selection

Models with all predictors always have the smallest RSS or largest R^2 on the *training set*. Therefore they are not suitable to choose the best one among models with different number of predictors.

We ought to *estimate the test error* on a test set. This may be done - indirectly by adjusting the training error to account for the bias due to overfitting: C_p (equivalently, AIC in case of linear model with Gaussian errors), BIC and adjusted R^2.

- Mallow's C_p:
$$C_p = \frac{1}{n}(RSS + 2d\hat{\sigma}^2)$$
where, d us the number of parameters and $\hat{\sigma}^2 \approx Var\epsilon$, typically estimated by using the *full model* containing all variables. C_p is an unbiased estimate of test MSE.

- AIC is defined for models fit by maximmum likelihood.
$$AIC = -2\log L + 2d$$
where, L is the maximum likelihood function for the estimated model. For linear regression with Gaussian error, $AIC \propto C_p$.

- BIC
$$BIC = \frac{1}{n}(RSS + \log(n)d\hat{\sigma}^2)$$
Since $\log n >$ for $n > 7$, the BIC places a higher penalty on models with many variables, and hence select smaller models than C_p.

- Adjusted R^2 (larger value is better)
$$\text{Adjusted } R^2 = 1 - \frac{RSS/(n-d-1)}{TSS/(n-1)}$$
$RSS/(n-d-1)$ may increase or decrease depends on d. Unlike R^2, adjusted R^2 pays a price for the inclusion of unnecessary variables in a model.

  C_p, AIC, BIC, adjusted R^2,are not appropriate in high-dimentional setting, as the estimated $\hat{\sigma}^2 \approx 0$ (when $p \geq n$).

- directly by cross-validation (or validation). It does not require estimate σ^2. It has a wide range of usage, as it may difficult to estimate d or σ^2. One can choose the model that has the smallest test error or using the *one-standard-error rule* to select the model that has a smaller size:
  - calculate the SE of the estimated test MSE for each model size.
  - identify the lowest test MSE
  - choose the smallest model for which its test MSE is within one SE of the lowest point.

6.4 Shrinkage methods for Variable selection

The shrinkage offers an alternative to selecting variables by adjusting a hyperparameter that trades-off RSS and the model parameter magnitudes. Shrinkage methods will produce a different set of coefficients for a different λ. Cross-validation may be used to select the best λ. After the λ is selected, one can fit a final model using the entire training data set.

The reason shrinkage methods may perform better than OLS is rooted in bias-variance trade-off: as λ increases, the flexibility of the model decreases b ecause of shrunk coefficients, leading to decreased variance but increased bias.

6.4.1 Ridge regression: minimize the following objective

$$RSS + \lambda \sum_{j=1}^{p} \beta_j^2$$

The ridge regression is equivalent to

$$\text{minimize } RSS \qquad \text{subject to } \sum_{j=1}^{p} \beta_j^2 \leq s$$

for some $s \geq 0$.

- It encourages the model parameters to shrink toward zero and find a balance between RSS and model parameter magnitudes. Cross-validation is used to find the best tuning parameter λ. When λ is large, $\beta_j \to 0$. Note, Ridge shrinks all coefficients and include all p variables.
- The OLS coefficients estimates are *scale equivariant*: regardless of how X_j is scaled, $X_j\hat{\beta}_j$ remain the same: if X_j is multiplied by c, this will simply leads to $\hat{\beta}_j$ be scaled by a factor of $1/c$.

- In contrast, when multiplying X_j by a factor, this may significantly change the ridge coefficients. Ridge coefficients depends on λ and the scale of X_j, and may even on the scaling of other predictors. Therefore, it is best practice to *standardize the predictors* before fitting a ridge model:

$$\tilde{x}_{ij} = \frac{x_{ij}}{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)}$$

- Ridge regression works best in situations where the OLS estimates have high variance, especially when p is large.
- Ridge will include all p variables in the final model.

6.4.2 The Lasso (Least Absolute Shrinkage and Selection Operator)

- The Lasso replaces the ℓ^2 error with ℓ^1 penalty. Lasso can force some coefficients to become exactly zero when λ is large enough. Thus it can actually performs *variable selection* hence better interpretation. Again, cross-validation is employed to select λ.
- The reason Lasso can perform variable selection is because the objective function is equivalent to

$$\text{minimizing RSS}, \text{subject to } \sum_{j=1}^{p} |\beta_j| \leq s$$

for some s. The contour of RSS in general only touch the ℓ_1 ball at its vertex, at which a minimum is obtained with some variables vanishes. In contrast, in the ridge situation, the ℓ_2 ball is round, and in general, the contour of the RSS function only touches the sphere at a surface point where a minimum is obtained with no variable vanishes.

- Neither ridge nor the lasso will universally dominate the other. When the response depends on a small number of predictors, one may expect lasso performs better; but in practice, this is never known in advance.
- Combining ridge and lasso leads to *elastic net* method.
- it is well known that ridge tends to give similar values coefficient values to correlated variables, while lasso may give quite different coefficient values to correlated variables.
- ridge regression shrinks all coefficients by the same proportion. While lasso perform **soft-thresholding**, shrink all coefficients by similar amount, and sufficient small coefficients are shrunk all the way to zero.

both ridge and lasso can be considered as computationally feasible approximation to the *best subset selection* which can be equivalently formulated as:

$$\text{minimize } RSS \qquad \text{subject to } \sum_{j=1}^{p} I(\beta_j \neq 0) \leq s.$$

- **Bayesian formulation**: Both ridge and lasso can be interpreted as maximize the **posterior probability** (MAP)

$$p(\beta|X,Y) \propto f(Y|X,\beta)p(\beta|X) = f(Y|X,\beta)p(\beta)$$

  where $p(\beta) = \prod_{j=1}^{p} g(\beta_j)$ with some density function g is the believed prior on β.

  - if g is Gaussian with mean zero and standard deviation a function of λ, then it follows the solution β given by the ridge is the same as maximizing the posterior $p(\beta|X,Y)$, that is, β is the posterior mode. In fact, β is also the posterior mean. Since the Gaussian prior is flat at near zero, ridge assumes the coefficients are randomly distributed about zero.
  - if g is double-exponential (Laplace) with mean zero and scale parameter a function of λ, then it follows the solution β given by the lasso is the same as maximizing the posterior $p(\beta|X,Y)$, that is, β is the posterior mode. In this case β is **not** the posterior mean. Since the Laplacian prior is steeply peaked at zero, lasso expects a priori that many coefficients are (exactly) zero.

6.5 Dimension reduction methods: transforming X_j.

There are two types of dimension reduction methods for regression: a) PCA regression, b) Partial list squares PLS. Both are designed to handle when the OLS breaks down due to that there are large number of correlated variables.

6.5.1 PCA regression: first use PCA to obtain M- PCA as linear combinations (directions) of the original p predictors:

$$Z_m = \sum_{j=1}^{p} \phi_{mj} X_j, \qquad 1 \leq m \leq M, \tag{6.1}$$

where, the ϕ_{mj} are called **PCA loadings**, and subject to the norm $\sum_{j=1}^{p} \phi_{mj}^2 = 1$ for each m. Note Z_m is a vector of length equal to the length of X_j, which is the number of data points n. The component of Z_m: z_{im}, $1 \leq i \leq n$ are called **PCA scores**. z_{im} is a *single number summary* of the p predictors with the m-th PCA for the i-th observation. PCA is not a feature selection method.

The first PCA defines the direction that contains the largest variance in X, and minimize the sum of squared perpendicular distances to each point (the projection error on the PCA), that is it defines the line that is *as close as possible* to the data; (In fact, the first PCA is given by the eigenvector of the largest eigenvalue of the covariance matrix $\frac{1}{n-1}X^TX$). The second PCA is orthogonal to the first PCA and has the second largest variance and is uncorrelated with the first, and so on. These directions are obtained in an *unsupervised way*, as Y is not used to obtain these components. Consequently, there is no guarantee that the directions that best explain the predictors will also be the best directions to use for predicting the response.

PCA is typically conducted after standardizing the data X, as without scaling, the high variance variables will tend to have higher influence on the obtained PCAs.

We then use OLS to fit a linear regression model

$$y_i = \theta_0 + \sum_{m=1}^{M} \theta_m z_{im} + \epsilon_i, \qquad i = 1, 2, \cdots, n \tag{6.2}$$

After substitute Equation 6.1 into equation Equation 6.2, one can find that

$$\sum_{m=1}^{M} \theta_m z_{im} = \sum_{j=1}^{p} \beta_j x_{ij}$$

with

$$\beta_j = \sum_{m=1}^{M} \theta_m \phi_{mj}. \tag{6.3}$$

Eq. Equation 6.3 has the potential to bias the coefficient estimates, but selecting $M << p$ can significantly reduce the variance. So model Equation 6.2 is a special case of linear regression subject to the constants Equation 6.3.

PCR and ridge are closely related and one can think of ridge regression as a continuous version of PCR.

6.5.2 Partial Least Squares

Similar to PCAR, PLS also first identifies a new set of features $Z_1, Z_2, \cdots, Z_m$, each of which is a linear combinations of the original features, and then fits a linear model via OLS with these new M features.

But PLS identifies these new features in a *supervised way*, that is, PLS uses Y in order to identify the new features that not only approximate the old features well, but also are *related to the response*, i.e., these new features explain both the response and the predictors.

First PLS standardizes the p predictors. PLS identifies the first component $Z_1 = \sum_{j=1}^{p} \phi_{1j} X_j$ by choosing $\phi_{1j} =< X_j, Y >$, the coefficient from the simple linear regression of Y onto X_j.

Since this coefficient is equal to the correlation between Y and X_j, PLS places the highest weight on the variables that are most strongly related to Y. The PLS direction does not fit the predictors as closely as does PCA, but it does a better job explaining the response.

Next, PLS orthogonality each X_j with respect to Z_1, that is, replace each X_j with the residual by regressing X_j on Z_1, and then repeat the same process.

When p is large, especially $p > n$, the forward selection method, shrinkage methods (lasso or ridge), PCR, PLR fit a *less flexible* model, hence particularly useful in performing regression in high-dimensional settings.

6.6 Homework:

- Conceptual: 1–4
- Applied: At least one.

6.7 Code Snippet

6.7.1 Python

```
np.isnan(Hitters['Salary']).sum()
```

6.7.2 Numpy

```
np.linalg.norm(beta_hat) #L2 norm. ord=1: L1  ord='inf': max norm.
```

6.7.3 Pandas

```
Hitters.dropna();
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas))
soln_path.index.name = 'negative log(lambda)'
```

6.7.4 Graphics

```
ax.errorbar(np.arange(n_steps),
            cv_mse.mean(1), #mean of each row (model)
            cv_mse.std(1) / np.sqrt(K), #estimate standard error of the mean
            label='Cross-validated',
            c='r') # color red

ax.axvline(-np.log(tuned_ridge.alpha_), c='k', ls='--') # plot a verticalline
```

6.7.5 ISLP and statsmodels

```
#Estimate Var(epsilon)

design = MS(Hitters.columns.drop('Salary')).fit(Hitters)
design.terms # to see the variable names in the design matrix
Y = np.array(Hitters['Salary'])
X = design.transform(Hitters)
sigma2 = OLS(Y,X).fit().scale  #.scale: RSE: residual standard error estimating

# Forward Selection using ISLP.models and a scoring function
from ISLP.models import \
     (Stepwise,
      sklearn_selected,
      sklearn_selection_path)
strategy = Stepwise.first_peak(design,
                               direction='forward',
                               max_terms=len(design.terms))
hitters_Cp = sklearn_selected(OLS,
                               strategy,
                               scoring=neg_Cp)
                               #default scoring MSE, will choose all variables
hitters_Cp.fit(Hitters, Y) # the same as hitters_Cp.fit(Hitters.drop('Salary', axis=1), Y)
hitters_Cp.selected_state_

#Forward selection using cross-validation
strategy = Stepwise.fixed_steps(design,
                                len(design.terms),
                                direction='forward')
full_path = sklearn_selection_path(OLS, strategy) #using default scoring MSE
full_path.fit(Hitters, Y) # there are , 19 variables, 20 models
Yhat_in = full_path.predict(Hitters)
```

```
#calculate in-sample mse

mse_fig, ax = subplots(figsize=(8,8))
insample_mse = ((Yhat_in - Y[:,None])**2).mean(0) #Y[:,None]: add a second axis, create a co
                        #[yw] mean(0): calculate mean along row, i.e., for each col. mean(1)

#Cross-validation
K = 5
kfold = skm.KFold(K,
                  random_state=0,
                  shuffle=True)
Yhat_cv = skm.cross_val_predict(full_path,
                                Hitters,
                                Y,
                                cv=kfold)
# Cross-validation mse
cv_mse = []
for train_idx, test_idx in kfold.split(Y):
    errors = (Yhat_cv[test_idx] - Y[test_idx,None])**2
    cv_mse.append(errors.mean(0)) # column means
cv_mse = np.array(cv_mse).T

#validation approach using ShuffleSplit
validation = skm.ShuffleSplit(n_splits=1, # only split one time.
                              test_size=0.2,
                              random_state=0)
for train_idx, test_idx in validation.split(Y):
    full_path.fit(Hitters.iloc[train_idx], #note needing to use iloc
                  Y[train_idx])
    Yhat_val = full_path.predict(Hitters.iloc[test_idx])
    errors = (Yhat_val - Y[test_idx,None])**2
    validation_mse = errors.mean(0)
```

6.7.6 sklearn

```
rom sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression

#Best subset selection using 10bnb
```

```
D = design.fit_transform(Hitters)
D = D.drop('intercept', axis=1) #needs to drop intercept
X = np.asarray(D)
path = fit_path(X,
                Y,
                max_nonzeros=X.shape[1]) #fit_path: a funciton from l0nb. use all variables
                # max_nonzeros: max nonzero coefficients in the fitted model.

# Ridge Regression
soln_array = skl.ElasticNet.path(Xs, # standardized, no intercept
                                 Y,
                                 l1_ratio=0., #ridge
                                 alphas=lambdas)
# Using pipline
ridge = skl.ElasticNet(alpha=lambdas[59], l1_ratio=0)
scaler = StandardScaler(with_mean=True,  with_std=True)
pipe = Pipeline(steps=[('scaler', scaler), ('ridge', ridge)])
pipe.fit(X, Y)
ridge.coef_

# Validation

validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.5,
                              random_state=0) # validation is a generator
ridge.alpha = 0.01
results = skm.cross_validate(ridge,
                             X,
                             Y,
                             scoring='neg_mean_squared_error',
                             cv=validation) # using the strategy defined in validation
-results['test_score']

# GridSearchCV()
param_grid = {'ridge__alpha': lambdas}
grid = skm.GridSearchCV(pipe,
                        param_grid,
                        cv=validation, # or use cv=kfold (5-fold CV defined separately)
                        scoring='neg_mean_squared_error') #default scoring=R^2
grid.fit(X, Y)
grid.best_params_['ridge__alpha']
grid.best_estimator_
grid.cv_results_['mean_test_score']
```

```
grid.cv_results_['std_test_score']


#Plot CV MSE
ridge_fig, ax = subplots(figsize=(8,8))
ax.errorbar(-np.log(lambdas),
            -grid.cv_results_['mean_test_score'],
            yerr=grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylim([50000,250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20);

# Use ElasticNetCV()
ridgeCV = skl.ElasticNetCV(alphas=lambdas, # ElasticNetCV accepts a sequence of alphas
                           l1_ratio=0,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler), # scaling is done once.
                         ('ridge', ridgeCV)])
pipeCV.fit(X, Y)
tuned_ridge = pipeCV.named_steps['ridge']
tuned_ridge.mse_path_
tuned_ridge.alpha_ # best alpha
tuned_ridge.coef_

# Evaluating test Error of Cross-validated Ridge

outer_valid = skm.ShuffleSplit(n_splits=1,
                               test_size=0.25,
                               random_state=1)
inner_cv = skm.KFold(n_splits=5,
                     shuffle=True,
                     random_state=2)
ridgeCV = skl.ElasticNetCV(alphas=lambdas, # a sequence of lambdas
                           l1_ratio=0,
                           cv=inner_cv) # K-fold validation
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgeCV)]);


results = skm.cross_validate(pipeCV,
                             X,
                             Y,
                             cv=outer_valid,
```

```
                              scoring='neg_mean_squared_error')
-results['test_score']

# Lasso regression
lassoCV = skl.ElasticNetCV(n_alphas=100, #test 100 alpha values
                           l1_ratio=1,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('lasso', lassoCV)])
pipeCV.fit(X, Y)
tuned_lasso = pipeCV.named_steps['lasso']
tuned_lasso.alpha_
tuned_lasso.coef_
np.min(tuned_lasso.mse_path_.mean(1)) # miminum avg mse

#to get the soln path
lambdas, soln_array = skl.Lasso.path(Xs, # standarsized, no -intercept
                                     Y,
                                     l1_ratio=1,
                                     n_alphas=100)[:2]

#PCA and PCR
pca = PCA(n_components=2)
linreg = skl.LinearRegression()
pipe = Pipeline([('scaler', scaler),
                 ('pca', pca),
                 ('linreg', linreg)])
pipe.fit(X, Y)
pipe.named_steps['linreg'].coef_
pipe.named_steps['pca'].explained_variance_ratio_

# perform Grid search
param_grid = {'pca__n_components': range(1, 20)} #PCA needs n_components >0
grid = skm.GridSearchCV(pipe,
                        param_grid,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y)

# cross-validation a null model
cv_null = skm.cross_validate(linreg,
                             Xn,
                             Y,
```

```
                            cv=kfold,
                            scoring='neg_mean_squared_error')
-cv_null['test_score'].mean()

#PLS
pls = PLSRegression(n_components=2,
                    scale=True) # standarsize the data
pls.fit(X, Y) # X has no-intercept

# Cross-validation
param_grid = {'n_components':range(1, 20)}
grid = skm.GridSearchCV(pls,
                        param_grid,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y)
```

7 Chapter 7: Moving Beyond Linearity

Often the linearity assumption of Y of X is good but it may have limited predictive power, as often a linear model is just an approximation. Imporovement can be obtained by reducing the model complexity (hence variance) with lasso, ridge, PCAR or PLR, etc. when it's not, - Polynomials - step functions - splines - local regression, - generalized additive models offer much more flexibility. All the above nonlinear methods falls into the *basis function* approach where we fit the following function

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \cdots + \beta_K b_K(x_i) + \epsilon_i$$

where the b_j, $j = 1, \cdots, K$ are pre-defined basis functions that transform X to a feature $b_j(X)$. Each approach corresponding to a choice of particular family of basis functions. The model can then fit with *OLS*.

7.1 Polynomials

The basis functions are simply the polynomial functions of different degrees. Polynomial terms of higher powers for X_j or interaction terms $X_i X_j$ are used, but the model is still a linear model in the coefficients β_j. The optimum degree d can be chosen by cross-validation. polynomial terms can be included in either a linear regression model or a logistic regression model.

In practice hardly a degree greater than 3 or 4 is used because a higher degree polynomial exhibits high degree of oscillation, especially near the boundary (Runge's phenomenon). This is because a polynomial imposes a *global structure*.

The standard error at a point $x_0$is calculated by

$$SE[\hat{f}(x_0)] = \ell_0^T \hat{\mathbf{C}} \ell_0$$

where, $\ell_0^T = (1, x_0, x_0^2, \cdots, x_0^d)$, and $\hat{\mathbf{C}}$ is the covariance matrix of the estimated coefficients β_j, $j = 0, 1, \cdots, d$ obtained from the OLS.

7.2 Step functions

A step function is a piece-wise constant function. Cut a X variable into $K+1$ regions using K cut points and then either use one-hot coding with $K+1$ dummy variables (and no intercept, in this case, each coefficient can be interpreted as the average value in that region) or create K dummy variables with an intercept to represent all those regions (in this case, the average value in that region equals to the intercept plus the coefficient). Choice of *cut-points* (knots) can be problematic. Binning the X variable amounts to convert a continuous variable into an *ordered categorical variable*.

The basis functions are simply *indicator functions* on each region:

$$b_j(x_i) = I(c_j \leq x_i < c_{j+1}).$$

7.3 Piececwise polynomials

It overcomes the disadvantage of polynomial basis which imposes a *global structure*. It fits separate low-degree polynomials over different regions of X separated by *knots*.

7.4 Splines

Splines are piece-wise polynomials of degree d that are continuous up to $d-1$ derivatives at each knot. E.g. a cubic spline with K knots has continuity up to second derivative at each knot, and has degree of freedom of $K+4$.

- linear spline: with knots ξ_k, $k = 1, \cdots, K$ is a piece-wise linear polynomial that is continuous at each knot.

  $$y = \beta_0 + \beta_1 b_1(x) + \cdots + \beta_{K+1} b_{K+1}(x) + \epsilon,$$

  where, b_k are **basis functions** defined by

  $$b_1(x) = x \tag{7.1}$$
  $$b_{k+1}(x) = (x - \xi_k)_+, \qquad k = 1, \cdots K \tag{7.2}$$

  Here the *positive part* is defined by

  $$x_+ = \begin{cases} x, & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- cubic splines: with knots ξ_k, $k = 1, \cdots, K$ is a piecewise cubic polynomials with continuous derivatives up to order 2 at each knot.

$$y = \beta_0 + \beta_1 b_1(x) + \cdots + \beta_{K+3} b_{K+3}(x) + \epsilon,$$

knot placement: General principle is that placing more knots in places where the function might vary most rapidly. In practice, it is common to place them at uniform quantiles of the observed X. This can be done by specifying a dof, and then let the algorithm to calculate the knots at uniform quantiles. Note a natural spline have more internal knots than a regression spline for the same dof.

For a **regular cubic spline**, the **basis functions** are defined by

$$b_1(x) = x \tag{7.3}$$
$$b_2(x) = x^2 \tag{7.4}$$
$$b_3(x) = x^3 \tag{7.5}$$
$$b_{k+3}(x) = (x - \xi_k)^3_+, \qquad k = 1, \cdots K \tag{7.6}$$

dof: $K + 4$ number of parameters (including the intercept). A regression spline can have high variance at the outer range of the predictors. To remedy this, one can use a *natural cubic spline*.

a **natural cubic spline** extrapolates linearly (as a linear function) beyond the internal knots. This adds 2×2 extra constrains. A natural cubic spline allows to put more internal knots for the same degree of freedom as a regular cubic spline.

dof: $K + 2$ (K only counts the internal knots; including the intercept).

For a **smoothing spline**: it is the solution g to the following problem:

$$\text{minimize}_{g \in S} \sum_{i=1}^{n} (y_i - g(x_i))^2 + \lambda \int g''(t) dt$$

The first term is the *loss* RSS and encourages $g(x_i)$ matches y_i. The second term is the *penalty* that penalize the *variability* in g (measured by $g''(t)$) by a tuning parameter $\lambda \geq 0$. If $\lambda = 0$ (no constraints on g), then the solution is just an interpolating polynomial. If $\lambda \to \infty$, then g is a linear function (because its second derivative is zero). λ controls the bias-variance trade-off.

The smoothing spline is in fact a natural spline with knots at unique values of x_i. But it is **different** than the natural spline. It is a *shrunk* version of a natural cubic spline, otherwise it would have too large (*nominal*) dof (number of parameters) because it has knots at unique values of x_i. It avoids the knot selection issue and leaving a single λ to tune. An *effective degrees of freedom* can be calculated for a smoothing spline as

$$df_\lambda = \sum_{i=1}^{n} \{\mathbf{S}_\lambda\}_{ii},$$

where $\mathbf{S}_\lambda$ is a $n \times n$ matrix determined by λ and x_i such that the vector of n fitted values can be written as

$$\hat{\mathbf{g}}_\lambda = \mathbf{S}_\lambda \mathbf{y}.$$

df_λ decreases from n to 2 as λ increases from 0 to ∞.

The LOO cross-validation error can be efficiently computed by

$$\text{RSS}_{cv}(\lambda) = \sum_{i=1}^{n} (y_i - \hat{g}_\lambda^{(-i)}(x_i))^2 = \sum_{i=1}^{n} \left[\frac{y_i - \hat{g}_\lambda(x_i)}{1 - \{\mathbf{S}_\lambda\}_{ii}} \right]^2$$

- Local Regression: a non-parametric method. It is similar to spline, but allowing regions overlap. With a sliding weight function of *span* s, fit separate (constant, linear, quadratic, for instance) fits over the range of X by weighted least squares.

The span s plays the similar role as λ in a smoothing spline, it controls the flexibility of the local regression. The smaller s is, the more *local and wiggle* will be the fit.

Local regression is a *memory based* procedure, because like KNN, all training data are needed each time when making a prediction.

Local regression can be generalized to *varying coefficient models* that fits a multiple linear regression model that is global in some variables but local in another, such as time.

Local regression can be naturally extends to p-dimension using a p-dimensional neighborhood, but really used when p is larger than 3 or 4 because there will be generally very few training examples near x_0 (curse of dimensionality)

- GAM (Generalized Additive Models): can be considered as an extension of multiple linear regression, replacing each feature $\beta_j X_j$ with an nonlinear function $f_j(X_j)$.

  $$y_i = \beta_0 + f_1(x_{i1}) + f_2(x_{i2}) + \cdots + f_p(x_{ip}) + \epsilon$$

  GAM can mix different f_j, for example, a spline, or a linear term or even include low order interactive terms. The coefficients are hard to interpret, but the fitted values are of interest.

GAM can be used in fitting a logistic regression model, that is

$$\log \frac{p(X)}{1 - p(X)} = \beta_0 + f_1(X_1) + f_2(X_2) + \cdots + f_p(X_p)$$

When fitting a GAM, and if OLS can not be used (such as when a smoothing spline is used), then the *back fitting* iterative method can be used: randomly initialize all variable coefficients; repeatedly hold all but one variable fixed, and perform a simple linear regression on that single variable, and update the corresponding coefficients until convergence. Convergence is typically very fast.

Pros and Cons of GAM

- flexible to model f_j, eliminating the need to try different transformations on each variable
- potentially more accurate prediction
- because the model is additive, can easily examine the effects of X_j on Y by holding all of the other variables fixed.
- The smoothness of f_j can be summarized by the effective dof.
- interaction terms $X_j X_k$ can be added.
- low dimensional interaction functions of the form $f_{jk}(X_j, X_k)$ can be added. Such term can be fit using a two-dimensional smoothers such as local regression or two dimensional splines.

7.5 Homework:

- Conceptual: 1–5
- Applied: At least one.

7.6 Code Snippet

7.6.1 Python

7.6.2 Numpy

```
Wage['education'].cat.categories # .cat is the categorical method accessor
Wage['education'].cat.codes
pd.crosstab(Wage['high_earn'], Wage['education'])

np.column_stack([Wage_['age'],
                         Wage_['year'],
                         Wage_['education'].cat.codes-1])

Xs = [ns_age.transform(age),
      ns_year.transform(Wage['year']),
      pd.get_dummies(Wage['education']).values] # -> 5 education levels: 1-hot coding
X_bh = np.hstack(Xs)
```

7.6.3 Pandas

```
cut_age = pd.qcut(age, 4) # cut based on the 25%, 50%, and 75% cutpoints. pd.cut is similar
```

7.6.4 Graphics

```
ax.legend(title='$\lambda$');
```

7.6.5 ISLP and statsmodels

7.6.6 sklearn

7.6.7 Useful code snippets

7.6.7.1 plot a model fit with confidence interval

```
def plot_wage_fit(age_df,
                  basis, # ISL model object
                  title):

    X = basis.transform(Wage)
    Xnew = basis.transform(age_df)
    M = sm.OLS(y, X).fit()
    preds = M.get_prediction(Xnew)
    bands = preds.conf_int(alpha=0.05)
    fig, ax = subplots(figsize=(8,8))
    ax.scatter(age,
               y,
               facecolor='gray',
               alpha=0.5)
    for val, ls in zip([preds.predicted_mean,
                        bands[:,0],
                        bands[:,1]],
                       ['b','r--','r--']):
        ax.plot(age_df.values, val, ls, linewidth=3)
    ax.set_title(title, fontsize=20)
    ax.set_xlabel('Age', fontsize=20)
    ax.set_ylabel('Wage', fontsize=20);
    return ax
```

7.6.7.2 Fitting with a step function

```
cut_age = pd.qcut(age, 4) # cut based on the 25%, 50%, and 75% cutpoints
# note pd.get_dummies(cut_age) is the X matrix
summarize(sm.OLS(y, pd.get_dummies(cut_age)).fit())
```

7.6.7.3 Fitting a spline

```
#specifying internal knots

bs_age = MS([bs('age',
                internal_knots=[25,40,60],
                name='bs(age)')]) #rename the variable names
Xbs = bs_age.fit_transform(Wage) # Xbs == bs_age above
M = sm.OLS(y, Xbs).fit()
summarize(M)

# specifying df
bs_age0 = MS([bs('age',
                 df=3, # df count does not include intercept. df=degree+ #knots
                 degree=0)]).fit(Wage)
Xbs0 = bs_age0.transform(Wage)
summarize(sm.OLS(y, Xbs0).fit())

BSpline(df=3, degree=0).fit(age).internal_knots_

# Fit a natural spline
ns_age = MS([ns('age', df=5)]).fit(Wage) #df=degree+ #knots -2
M_ns = sm.OLS(y, ns_age.transform(Wage)).fit()
summarize(M_ns)

# fit a smoothing spline
X_age = np.asarray(age).reshape((-1,1))
gam = LinearGAM(s_gam(0, lam=0.6)) #gam is the smoothing spline model with a given lambda
gam.fit(X_age, y)

#Fiting a smoothing spline with an optimized lambda
gam_opt = gam.gridsearch(X_age, y)


# Fitting a smoothin spline by specifying a df (not including intercept)
fig, ax = subplots(figsize=(8,8))
```

```
ax.scatter(X_age,
           y,
           facecolor='gray',
           alpha=0.3)
for df in [1,3,4,8,15]:
    lam = approx_lam(X_age, age_term, df+1) # find the lambda corresponding to a df.
    age_term.lam = lam # update lambda
    gam.fit(X_age, y)
    ax.plot(age_grid,
            gam.predict(age_grid),
            label='{:d}'.format(df),
            linewidth=4)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Wage', fontsize=20);
ax.legend(title='Degrees of freedom');
```

7.6.8 GAM

```
### manually contruct basis
ns_age = NaturalSpline(df=4).fit(age) #df counts do not include intercepts. -> 4 columns
ns_year = NaturalSpline(df=5).fit(Wage['year']) # -> 5 cols
Xs = [ns_age.transform(age),
      ns_year.transform(Wage['year']),
      pd.get_dummies(Wage['education']).values] # -> 5 education levels: 1-hot coding
X_bh = np.hstack(Xs)
gam_bh = sm.OLS(y, X_bh).fit()

### Examinge partial effect

age_grid = np.linspace(age.min(),
                       age.max(),
                       100)
X_age_bh = X_bh.copy()[:100] # Take the first 100 rows of X_bh
# calculate the row mean and make it a row vector in the shape of 1Xp, then broadcast
X_age_bh[:] = X_bh[:].mean(0)[None,:]
X_age_bh[:,:4] = ns_age.transform(age_grid)# replace the first 4 cols with basis functions ev
preds = gam_bh.get_prediction(X_age_bh) #gam_bh is the GAM model with all 14 basis
bounds_age = preds.conf_int(alpha=0.05)
partial_age = preds.predicted_mean
```

```
center = partial_age.mean() # center of the prediction
partial_age -= center # center the prediction for better viz
bounds_age -= center
fig, ax = subplots(figsize=(8,8))
ax.plot(age_grid, partial_age, 'b', linewidth=3)
ax.plot(age_grid, bounds_age[:,0], 'r--', linewidth=3)
ax.plot(age_grid, bounds_age[:,1], 'r--', linewidth=3)
ax.set_xlabel('Age')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of age on wage', fontsize=20);


### Using a smoothing spline and pygam package
#### Specifying lambda
#### default \lambda = 0.6 is used.
gam_full = LinearGAM(s_gam(0) + # spline smoothing applies to the first col of the feature ma
                     s_gam(1, n_splines=7) + # smoothing applied to the 2nd col
                     f_gam(2, lam=0)) # smothing applied to the 3rd col: a factor col
Xgam = np.column_stack([age,  #stack as columns
                        Wage['year'],
                        Wage['education'].cat.codes])
gam_full = gam_full.fit(Xgam, y)

gam_full.summary() # verbose summary

#### Plot partial effect using a plot_gam from ISLP.pygam
fig, ax = subplots(figsize=(8,8))
plot_gam(gam_full, 0, ax=ax) # 0: partial plot of the first component: age
ax.set_xlabel('Age')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of age on wage - default lam=0.6', fontsize=20);

### Specifying df
age_term = gam_full.terms[0]
age_term.lam = approx_lam(Xgam, age_term, df=4+1)
year_term = gam_full.terms[1]
year_term.lam = approx_lam(Xgam, year_term, df=4+1)
gam_full = gam_full.fit(Xgam, y)

#### Plot partial effect
fig, ax = subplots(figsize=(8, 8))
ax = plot_gam(gam_full, 2)
ax.set_xlabel('Education')
```

```
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of wage on education',
             fontsize=20);
ax.set_xticklabels(Wage['education'].cat.categories, fontsize=8);
```

7.6.8.1 Anova for GAM

```
gam_0 = LinearGAM(age_term + f_gam(2, lam=0)) # note age_term is a s_gam with df=4 defined al
gam_0.fit(Xgam, y)
gam_linear = LinearGAM(age_term +
                       l_gam(1, lam=0) +
                       f_gam(2, lam=0))
gam_linear.fit(Xgam, y)
anova_gam(gam_0, gam_linear, gam_full)
```

7.6.8.2 Logistic GAM

```
gam_logit = LogisticGAM(age_term +
                        l_gam(1, lam=0) +
                        f_gam(2, lam=0))
gam_logit.fit(Xgam, high_earn)
```

7.6.8.3 LOESS

```
lowess = sm.nonparametric.lowess
fig, ax = subplots(figsize=(8,8))
ax.scatter(age, y, facecolor='gray', alpha=0.5)
for span in [0.2, 0.5]:
    fitted = lowess(y,
                    age,
                    frac=span,
                    xvals=age_grid)
    ax.plot(age_grid,
            fitted,
            label='{:.1f}'.format(span),
            linewidth=4)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Wage', fontsize=20);
ax.legend(title='span', fontsize=15);
```

8 Chapter 8: Tree-Based Methods

(Decision) tree-based methods stratify or segment the predictor space into a number of simple regions using tree-based rules. The predictor space is subdivided into distinct and non-overlapping high-dimensional boxes along each axis. Such methods are simple and easy for interpretation. Their predicting accuracy is not as good as the best supervised learning approaches. However, through **ensemble method** such as *bagging, random forests, boosting, Bayesian additive regression trees*, by growing and combining large number of trees (weak learners) to yield a single consensus prediction may result in dramatic improvement in prediction accuracy, at the expense of some loss in interpretation. Decision trees often overfit the training data: a small change in the date might cause a large change in the tree. A small tree may offer small variance and better interpretation. Tree can easily handle a categorical variable without creating dummy variables.

A decision tree is typically drawn *upside down*. An *internal node* is a point along the tree where the predictor space is split, and a *terminal node* (leaf node) is a point along the tree that do not split. These terminal nodes are the final split regions of the predictor space. A *branch* connects nodes.

Decision trees can be applied to both regression and classification problems.

8.1 Regression tree

For a Regression tree, The value at a leaf node equals to the average of the Y values of all examples in the leaf node. The objective is to minimize the RSS

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2 \tag{8.1}$$

where, $\hat{y}_{R_j}$ is the mean response for the training observations in the j-th box R_j that corresponds to the j-th leaf node.

The first node (root) is the most important predictor, and so on.

It is computationally intractable to consider every possible partition of the feature space into J boxes in objective Equation 8.1. The solution is to use a *top-down* and *greedy* **recursive binary splitting**. It is greedy (myopic) because at each step of the tree-building process, the

best split is decided at that particular step by choosing a predictor X_j (consider all predictors) and a cut-point s (consider all values of that predictor) that leads to the greatest reduction in RSS, rather than looking ahead and picking a split that will lead to a better tree in some future step. The greedy splitting amounts to at each step, only the current region is split by a feature. This process is repeated within each of the resulting regions, until a stopping criterion.

Stopping criterion: - max number of observations in each leaf - max number of depth - RSS decrease smaller than a threshold.

8.2 Classificaiton tree

For a Classification tree: An example is classified as the class which is the mode of the examples in the leaf node. The training objective is similar to Equation 8.1, but with RSS replaced with

- classification error rate $E = 1-\max_k(\hat{p}_{mk})$, where $\hat{p}_{mk}$ is the fraction of training examples in the mth region that are in the k-th class. But this measure is not sufficiently sensitive for tree-growing (node-splitting).
- Gini index that measures node purity (total variance): $G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$. The Gini index takes on small value if all of the $\hat{p}_{mk}$'s are close to zero or one, indicating that a node contains predominantly observations from a single class.
- Cross-entropy: very similar to Gini index numerically, $D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$. Cross-entropy is always non-negative. Cross entropy is the expectation of the information contained in a probability information.

Gini index and Cross-entropy are preferred when splitting a node, while Classification error rate is preferred when pruning a tree if the prediction accuracy is the final goal.

Two leaf nodes might have the same predicted value resulting from a split, this is because the two leaf nodes have different node purity, which amounts to the certainty of a predicted value. In this case, an observation falls into the leaf node with higher purity renders higher certainty.

8.3 Prunning a tree

Using stopping criteria directly to obtain a small tree may be near-sighted: A seemingly worthless spit early on might be followed by a very good split with large RSS reduction. A better approach is to grow a very large tree T_0 such that each leaf only has some minimum number of observations, and then **prune** it back in order to obtain a *subtree* with the least

test error via cross-validation or validation approach. *Cost complexity pruning* (weakest link pruning) is used to do this by minimizing the following with a tuning α:

$$\sum_{m=1}^{|T_\alpha|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T_\alpha|$$

where, $|T_alpha|$ is the number of leaf nodes in T_α, which is the best subtree that minimize the above objective. As we increase α from zero, branches get pruned from the tree in a nested and predictable fashion, so obtaining the sequence of T_α is easy. The formulation is similar to lasso. And then An optimal $\hat{\alpha}$ is chosen by cross-validation, and the corresponding $T_{\hat{\alpha}}$.

8.4 Bagging

Bootstrapping aggregation, or *bagging* is a general purpose procedure for reducing variance of a statistical learning method because of the Law of Large Numbers: given a set of n independent observations $Z_1, \cdots, Z_n$ with variance σ^2, the variance of the mean $\bar{Z}$ is σ^2/n. In other words, averaging a set of observations reduces variance.

But in practice, we typically do not have access to multiple training sets. Instead, we **bootstrap** the training set to obtain B (usually hundreds or even thousands) bootstrapped training sets, and then fit a separate tree (usually deep and not pruned, hence with low bias) independently for the b-th bootstrapped training set to get the prediction $\hat{f}^{*b}(x)$ at x for each b, and then finally combine all the trees by averaging all the predictions to obtain

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x).$$

The above formula works for regression. For classification, the average is replaced by *majority vote*. Using large B in bagging (including RF) typically does not lead to overfitting. But small B may underfit. Bagging often leads to correlated (similar) trees, and can get caught in local optima and thus fail to explore the model space, and thus averaging may not lead to large reduction in variance. One way to remedy this is by RF.

8.4.1 Out-of-Bag Error Estimate

On average, each bagged tree makes use of 2/3 of the total observations. The remaining 1/3 of the observations not used to fit a given bagged tree are referred to as the *out-of-bag* (OOB) observations. For each observation, it is an OOB observation in around $B/3$ trees, and hence the average of the predictions of those $B/3$ trees for the i-the observation can be used as a cross-validation error for observation i. The overall OOB error can be calculated this way for all n observations.

When B is large, such as $B = 3n$, then this is essentially the LOO cross-validation error for bagging. This is cheap way to evaluate test error without the need of cross-validation (which may be onerous) or validation approach.

8.4.2 Random Forests

Bagging results in correlated trees and thus the variance may not be reduced by the average. Random forests still grows independent trees using bootstrapped data sets of the original data set, but RF *decorrelates* the trees by *randomly selecting* m predictors ($m < p$) each time a split in a tree is considered. Typically $m \approx \sqrt{p}$. Thereby leading to a more thorough exploration of model space. Bagging is the case when $m = p$. On average, $(p - m)/p$ of the splits will not consider a specific predictor. Using a small value of m in building RF will typically helpful when there are a large number of correlated predictors.

Large B will not lead to RF to overfit, but small B may underfit.

8.4.3 Variable Importance Measure (VI)

For bagged/RF regression trees, VI is the total amount of RSS decreased due to splits over a given predictor, averaged over all B trees. A large VI value indicates an important predictor. For bagged/RF classification trees, replacing RSS with Gini index or cross-entorpy.

8.5 Boosting

Like Bagging, boosting is a general approach that can be applied to many statistical learning methods for regression and classification. Boosting does not involve bootstrap sampling; Boosting grows trees *sequentially* by **slow** learning: each new tree is grown by fitting a new tree to the residuals (modified version of the original data set) left over from the previous trees, and then a shrunken version of the new tree is added to the model. Each new tree tries to capture signal that is not yet accounted for by the current set of trees.

8.5.1 Boosting Algorithm for regression trees

1. Set $\hat{f}(x) = 0$, and $r_i = y_i$ for all i in the training set.
2. For $b = 1, \cdots, B$, repeat 2.1. Fit a tree $\hat{F}^b$ with d splits ($d+1$ terminal nodes, can involve at most d variables) to the training data (X, r). 2.2. Update $\hat{(f)}$ by adding a shrunken version of the new tree:
$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x).$$

2.3. Update the residuals

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$$

3. Output the boosted model

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x).$$

Each new tree can be rather small ($d = 1$ or 2, hence with low variance) controlled by the parameter d. By fitting a small tree to the residual, we slowly improve $\hat{f}$ in areas where it does not perform well. The shrinkage λ slows the learning, allowing more and different shaped trees to attack the residuals.

8.5.2 Tuning parameters for Boosting

- The number of trees B: unlike bagging and random forests, boosting can overfit if B is too large, although overfitting tends to occur slowly. B is selected with cross-validation.
- The shrinkage parameter λ: Typical values are 0.01 or 0.001. Very small λ may require very large B.
- The number of splits d: d controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a *stump*, consisting of a single split. In this case, the boosted ensemble is fitting an additive model, since each term involves only one variable, hence easy to interpret. d is the *interaction depth*, controls the interaction order of the boosted model, since d splits can involve at most d variables.

8.6 Bayesian Additive Regression Trees (BART)

BART is related to the approaches used by both bagging and boosting. We only make use of the original data (not using bootstrap) and their modified version (residuals from other trees) , and grow trees sequentially.

- each tree tries to capture the signal not yet accounted by for by the current model, as in boosting.
- each tree is constructed in a random manner as in bagging and RF

The main novelty of BART is the way in which new trees are generated. Assume there are K trees and B iterations. Let $\hat{f}_k^b(x)$ be the prediction at x for the kth tree used in the bth iteration.

Initially, BART initializes all trees to be a single root node, with $\hat{f}_k^1(x) = \frac{1}{nK}\sum_{i=1}^{n} y_i$. Thus $\hat{f}^1(x) = \sum_{k=1}^{K} \hat{f}_k^1(x) = \frac{1}{n}\sum_{i=1}^{n} y_i$.

In subsequent iteration, BART updates each of the K trees, one at a time. In the b-th iteration, to update the kth tree, obtain a *partial residual* by subtracting from each response y_i the predictions from all but the k-th tree,

$$r_i = y_i - \sum_{k'<k} \widehat{f}^b_{k'}(x_i) - \sum_{k'>k} \widehat{f}^{b-1}_{k'}(x_i)$$

for each observation $i = 1, \cdots, n$. Note when $k' < k$, the trees are updated already in the b-th iteration, and for $k' > k$, the trees are from the previous iteration $b-1$. Rather than fitting a fresh tree to this partial residual r_i, BART obtain a new tree $\widehat{f}^b_k$ by *randomly perturb* the tree $\widehat{f}^{b-1}_k$ from the $(b-1)$-th iteration via the following operations:

1. change the structure of $\widehat{f}^{b-1}_k$ by adding or pruning branches
2. keep the same structure of $\widehat{f}^{b-1}_k$ but perturb the prediction values.

Perturbations that improve the fit are favored. The perturbation only modifies the previous tree slightly hence guard against overfitting. In addition, the size of each tree is limited to avoid overfitting. The perturbation can be interpreted as drawing a new tree from a *posterior* distribution via *Markov chain Monte Carlo* sampling. The perturbation avoids local minima and achieve a more thorough exploration of the model space.

At the end of each iteration, the K trees from that iteration will be summed, i.e. $\widehat{f}^b(x) = \sum_{k=1}^K \widehat{f}^b_k(x)$ for $b = 1, \cdots, B$.

Finally, computer the mean (or other quantities such as percentile) after L burn-in samples:

$$\widehat{f}(x) = \frac{1}{B-L} \sum_{b=L+1}^{B} \widehat{f}^b(x).$$

During the *burn-in* period- the first L iterations, $\widehat{f}^\ell$, $\ell <= L$ tends not to provide good results, hence are discarded.

BART has very impressive out-of-box performance: perform well (not overfitting) with minimal tuning.

Parameters:

- number of trees, K: e.g., $K = 200$
- number of iterations: B: e.g., $B = 1000$
- burn-in iterations L: e.g., $L = 100$.

8.7 Homework:

- Conceptual: 1–6
- Applied: At least one.

8.8 Code Snippet

8.8.1 Python

8.8.2 Numpy

```
np.asarray(D)
```

8.8.3 Pandas

```
feature_imp.sort_values(by='importance', ascending=False)
```

8.8.4 Graphics

8.8.5 ISLP and statsmodels

8.8.6 sklearn

8.8.7 Useful code snippets

8.8.7.1 Classification Decision Tree

```
from sklearn.metrics import (accuracy_score,
                             log_loss)

clf = DTC(criterion='entropy',
          max_depth=3,
          random_state=0)
clf.fit(X, High)
accuracy_score(High, clf.predict(X))
resid_dev = log_loss(High, clf.predict_proba(X))
ax = subplots(figsize=(12,12))[1]
```

```
plot_tree(clf,
          feature_names=feature_names,
          ax=ax);
print(export_text(clf,
                  feature_names=feature_names,
                  show_weights=True))

# Using validaiton approach to train and test the model
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=200,
                              random_state=0)
results = skm.cross_validate(clf,
                             D,
                             High,
                             cv=validation)
results['test_score']
```

8.8.7.2 Pruning a classifcaiton Decision tree

```
(X_train,
 X_test,
 High_train,
 High_test) = skm.train_test_split(X,
                                   High,
                                   test_size=0.5,
                                   random_state=0)
clf = DTC(criterion='entropy', random_state=0)
clf.fit(X_train, High_train)
accuracy_score(High_test, clf.predict(X_test))
ccp_path = clf.cost_complexity_pruning_path(X_train, High_train)
kfold = skm.KFold(10,
                  random_state=1,
                  shuffle=True)
grid = skm.GridSearchCV(clf,
                        {'ccp_alpha': ccp_path.ccp_alphas},
                        refit=True, # Refit the best estimator with the entire dataset
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X_train, High_train)
grid.best_score_
best_ = grid.best_estimator_
best_.tree_.n_leaves
```

```
print(accuracy_score(High_test,
                     best_.predict(X_test)))
confusion = confusion_table(best_.predict(X_test),
                            High_test)
```

8.8.7.3 Fitting a regression tree

```
reg = DTR(max_depth=3)
reg.fit(X_train, y_train)
ax = subplots(figsize=(12,12))[1]
plot_tree(reg,
          feature_names=feature_names,
          ax=ax);
```

8.8.7.4 Pruning a regression tree

```
ccp_path = reg.cost_complexity_pruning_path(X_train, y_train)
kfold = skm.KFold(5,
                  shuffle=True,
                  random_state=10)
grid = skm.GridSearchCV(reg,
                        {'ccp_alpha': ccp_path.ccp_alphas},
                        refit=True,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
G = grid.fit(X_train, y_train)
best_ = grid.best_estimator_
np.mean((y_test - best_.predict(X_test))**2)
```

8.8.7.5 Bagging and RG

```
bag_boston = RF(max_features=X_train.shape[1], random_state=0, n_estimators=500)
bag_boston.fit(X_train, y_train)
y_hat_bag = bag_boston.predict(X_test)


#RF
RF_boston = RF(max_features=6,
               random_state=0).fit(X_train, y_train)
y_hat_RF = RF_boston.predict(X_test)
np.mean((y_test - y_hat_RF)**2)
```

```
#VI
feature_imp = pd.DataFrame(
    {'importance':RF_boston.feature_importances_},
    index=feature_names)
feature_imp.sort_values(by='importance', ascending=False)
```

8.8.7.6 Gradient Boosting

```
boost_boston = GBR(n_estimators=5000,
                   learning_rate=0.001,
                   max_depth=3,
                   random_state=0)
boost_boston.fit(X_train, y_train)
test_error = np.zeros_like(boost_boston.train_score_)
for idx, y_ in enumerate(boost_boston.staged_predict(X_test)):
   test_error[idx] = np.mean((y_test - y_)**2)

plot_idx = np.arange(boost_boston.train_score_.shape[0])
ax = subplots(figsize=(8,8))[1]
ax.plot(plot_idx,
        boost_boston.train_score_,
        'b',
        label='Training')
ax.plot(plot_idx,
        test_error,
        'r',
        label='Test')
ax.legend();
```

8.8.7.7 BART

```
bart_boston = BART(random_state=0, burnin=5, ndraw=15) #num_trees=200, max_states=100
# ndraw: number of iterations or samples to draw from the posterior distribution after the bu
bart_boston.fit(X_train, y_train)
yhat_test = bart_boston.predict(X_test.astype(np.float32))
np.mean((y_test - yhat_test)**2)

# Variable Inclusion
var_inclusion = pd.Series(bart_boston.variable_inclusion_.mean(0),
                                index=D.columns)
```

9 Chapter 9: Support Vector Machine

Idea: To attak a two-class classification problem directly: *try and find a (hyper)plane that separates the classes in the feature space.*

If we can not, we relax the conditions: - soften what we mean by "separating" by allowing misclassified points (Support Vector Classifier) - enrich and enlarge the feature space so that the separation is possible with non-linear decision boundary.

9.1 What is a hyperplane?

A hyperplane is defined by the following linear equation

$$\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p = 0.$$

It is a $p-1$ dimension **flat affine** subspace (affines means not necessarily pass the origin). - When $p = 2$, it is a line. - When β_0, it passes through the origin, becomes a *subspace*. - The normal vector $\beta = (\beta_1, \beta_2, \cdots, \beta_p)$ is perpendicular to the hyperplane. - Let $f(X) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$, then $f(X) = 0$ defines the hyperplane which separte the space into two halves, and for points on one side of the hyperplane, $f(X) > 0$, and vice versa. - If we code $Y_i = 1$ for $f(X_i) > 0$, and $Y_i = -1$ for $f(X_i) < 0$, then we always have

$$Y_i \cdot f(X_i) > 0 \qquad \text{for all } i.$$

If $f(x^*)$ is far from zero, then we are more confident that the test point x^* belongs to a class.

9.2 Maximal Margin Classifier (Optimal Seperating Hyperplane)

When the data can be perfectly separated using a hyperplane, among all infinitely many separating hyperplanes, the maximal margin classifier makes the biggest gap or margin between two classes. It is the solution of the following convex quadratic program

$$\text{maximize}_{\beta_0, \beta_1, \cdots, \beta_p} M$$

$$\text{subject to } \sum_{j=1}^{p} {\beta_j}^2 = 1 \qquad \text{and} \qquad y_i(\beta_0 + \beta_1 x_{xi} + \cdots + \beta_p x_{ip}) \geq M \text{ for all } i = 1, \cdots, n$$

The constraints guarantees that each observation will be on the correct side of the hyperplane. The normalizing constraint allows to interpret

$$y_i(\beta_0 + \beta_1 x_{xi} + \cdots + \beta_p x_{ip})$$

to be the perpendicular distance from observation i to the hyperplane. Hence M represents the *margin* of our hyperplane. In a sense, the maximal margin hyperplane represents the mid-line of the widest "slab" that we can insert between the two classes. Maximal margin classifier may lead to overfitting when p is large.

The observations that lie along the margin are called *support vectors*. They affect the optimal separating hyperplane. Other observations that outside the separating margin do not affect the optimal separating plane, provided their movement do not cause them to cross the boundary set by the margin.

9.3 Support Vector Classifier (soft margin classifier)

Often time: - the data is not separable by a linear plane, hence there is no maximal margin classifier. - or the data is noisy, and a poor maximal margin separating plane is obtained. leading to a classifier that is sensitive to a single observation (overfitting).

A support Vector Classifier maximizes a *soft* margin to almost separates the classes:

$$\text{maximize}_{\beta_0,\beta_1,\cdots,\beta_p} M, \text{ subject to } \sum_{j=1}^{p} {\beta_j}^2 = 1$$

and

$$y_i(\beta_0 + \beta_1 x_{xi} + \cdots + \beta_p x_{ip}) \geq M(1 - \epsilon_i), \text{ where } \epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C. \tag{9.1}$$

The soft margin may be violated by some observations. ϵ_i are *slack varaibles* that individual observations to be on the wrong side of the margin or the hyperplane. - If $\epsilon_i = 0$, then the i-th observation is on the correct side of the margin. - If $1 > \epsilon > 0$, then the i-th observation is on the wrong side of the margin. - If $\epsilon > 1$, then it is on the wrong side of the hyperplane.

C is a regulation parameter that can be tuned with cross-validation, bounding the sum of ϵ_i's, i.e., it determines the number and severity of the violations to the margin (and to the hyperplane) that we will tolerate. So C is a *budget* for such violations. - If $C = 0$, then all $\epsilon_i = 0$ for each i, and the support vector classifier becomes the maximal margin hyperplane. - If $C > 0$, then no more than C observation can be on the wrong side of the hyperplane then $\epsilon_i > 1$. As C increases, more tolerant to the violations leading to *wider* margin, and more support vectors.

So C controls the bias-variance trade-off - When C is small, less tolerance and smaller margin, the classifier may highly fit the data, hence small bias but high variance.

Similar to the maximal margin classfier, the support vector classfier is only affected by the *support vector* points on the margin or that violate the margin, robust to the points that are far away from the hyperplane. This is in contrast to some other classifiers such as LDA which needs a class mean of all within class observations, and a within-class covariance computed using *all* observations.

On the other hand, support vector classifer is very similar to Logistic regression, which is also not sensitive to observations far from the decision boundary.

9.4 Feature (Basis) Expansion for nonlinear decision boundary

Sometimes, a linear boundary can fail no matter what C takes on. One way to fix this is to enlarge the feature space by including transformations such as $X_1^2, X_1^3, X_1X_2, X_1X_2^2, \cdots$. Hence go from a p dimension space to a higher dimension space. This results in non-linear decision boundaries in the original space. For example, the adding of $X_1, X_2, X_1^2, X_2^2X_1X2$ would have decision boundary of the form

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_2^2 + \beta_5 X_1 X_2 = 0.$$

9.5 Kennel trick and Support Vector Machines

The kernel trick is simply an efficient computational approach that enacting enlarging the feature space. The linear support vector classifier can be represented by

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle$$

The parameters α_i can be estimated on a training set by computing the $\binom{n}{2}$ inner products between pairwise training examples $\langle x_i, x_i' \rangle$. It turns out most $\hat{\alpha}_i$ are zeros, and

$$f(x) = \beta_0 + \sum_{i \in S} \hat{\alpha}_i \langle x, x_i \rangle$$

Where S is the *support set* of indices i such that $\hat{\alpha}_i > 0$. Note all $\hat{\alpha}_i \geq 0$. The inner product $\langle x, x_i \rangle$ can be rewritten as a *linear* kernel (linear on the feature x)

$$K(x, x_i) = \langle x, x_i \rangle$$

thus the linear support vector classifier becomes

$$f(x) = \beta_0 + \sum_{i \in S} \hat{\alpha}_i K(x, x_i).$$

The linear kernal function essentially quantifies the **similarity** of a pair of observations using **Pearson** (standard) correlation. Generalize this idea, one could use other form of kernels to measure the similarity.

For a nonlinear boundary determined by a polynomial of degree d feature space with p variables, the kernel function is given by

$$K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij} x_{i'j}\right)^d$$

When the support vector classifier is combined with a non-linear kernel, the resulting classifier is a **support vector machine (SVM)**. It essentially fits a support vector classifier in a higher dimensional space involving polynomials of degree d.

The kernel function will allow easy computation for the inner product of the $\binom{p+d}{d}$ monomial basis functions without explicitly working in the enlarged feature space. This is important because in many applications, the enlarged feature space is large so that the computations are intractable. For some other kernel such as *radial kernel*, the feature space is *implicit* and infinite-dimensional.

A quick **proof** of the fact that there are $\binom{p+d}{d}$ monomial basis funtions for the space of polynomials of degree d in p variables.

1. Use the *stars and bars* method, it is easy to see that for a fixed degree δ, there are

   $$\binom{p+\delta-1}{p-1} = \binom{p+\delta-1}{\delta}$$

   monomials. This can be understood as distributing $p-1$ bars separating δ starts (each representing one degree) into p bins, each bin representing a variable.
2. Adding the number of monomials for $\delta = 0, 1, \cdots, d$,

   $$\binom{p+0-1}{0} + \binom{p+1-1}{1} + \cdots + \binom{p+d-1}{d}$$

   Rewrite $\binom{p+0-1}{0}$ as $\binom{p+1-1}{0}$ and apply the Pascal formula

   $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n}{k-1}$$

   repeatedly, to obtain the sum is $\binom{p+d}{d}$.

A final note is that SVM can also be used for regression, known as *support vector regression*, in which SVR seeks coefficents (β_j) that minimize the a loss where only residuals larger in absolute value than some positive constant contributes to the loss.

Other Common Used Kernels - Radial Kernel

$$K(x_i, x_i') = \exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2)$$

When a test point x^* is far from a training example x_i, then the kernel function value is small and plays little role in $f(x^*)$. So radial kernel has a *local behavior*: in the sense that only nearby points have an effect on the class label of a test observation.

As γ increases (fewer local training points are included in a decision), the fit becomes more non-linear (local) and the training error decreases.

How to apply SVM to multiple class classification

- OVA (one-vs-rest): One vs All: Fit K different 2-class SVM classifiers $\hat{f}_k(x)$, $k = 1, \cdots, K$. Classify x^* to the class for which $\hat{f}_k(x^*)$ is the largest, as this amounts to a high level of confidence that the test observation belogs to.
- OVO (all-pairs): One vs. One: Fit all $\binom{K}{2}$ pairwise classifiers $\hat{f}_{k\ell}(x)$. Classify x^* to the class that wins the most pairwise competitions.

If K is not too large, use OVO.

9.6 SVM vs. Logistic Regression

SVM can be viewed as minimizing the following *hinge loss*

$$\text{minimize}_{\beta_0, \beta_1, \cdots, \beta_p} \left\{ \sum_{i=1}^{n} \max[0, 1 - y_i f(x_i)] + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

When λ is large, then β_j's are small, more violations to the margin are tolerated, and a low-variance and high-bias classier will result. A small value of λ amounts to a small value of C in Equation 9.1.

The hinge loss function is very similar to the negative log-likelihood loss for the logistic regression. The loss function $\text{minimize}_{\beta_0, \beta_1, \cdots, \beta_p} \sum_{i=1}^{n} \max[0, 1 - y_i f(x_i)]$ is zero when $y_i(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}) \geq 1$; Theses corresponds to when an observation in **on the correct side of the margin**. In contrast, the loss function for logistic equation is not exactly zero anywhere, but it is very small for observations that are far from the decision boundary.

- When classes are nearly separable, SVM does better than LR, so does LDA. When not, LR is preferred.
- when not, LR with ridge penalty and SVM are very similar.
- If wish to estimate probability, then LR is the choice.
- For nonlinear boundary, kernel SVMs are popular. Can use kernels with LR and LDA as well, but computations are more expensive.

9.7 Homework:

- Conceptual: 1–3
- Applied: At least one.

9.8 Code Snippet

9.8.1 Python

```
roc_curve = RocCurveDisplay.from_estimator # shorthand for the function from_estimator()
```

9.8.2 Numpy

9.8.3 Pandas

9.8.4 Graphics

```
from matplotlib.pyplot import subplots, cm #cm for colormap
ax.scatter(X[:,0],
           X[:,1],
           c=y,
           cmap=cm.coolwarm);
```

9.8.5 ISLP and statsmodels

9.8.6 sklearn

9.8.7 Useful code snippets

9.8.7.1 SVM

```
svm_linear_small = SVC(C=0.1, kernel='linear')
svm_linear_small.fit(X, y)
fig, ax = subplots(figsize=(8,8))
plot_svm(X,
         y,
         svm_linear_small,
         ax=ax)
svm_linear.coef_
svm_linear.intercept_

### Tuning a parameter
kfold = skm.KFold(5,
                  random_state=0,
                  shuffle=True)
grid = skm.GridSearchCV(svm_linear,
                        {'C':[0.001,0.01,0.1,1,5,10,100]}, # 7 values
                        refit=True,
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X, y)
grid.best_params_
grid.cv_results_
grid.cv_results_[('mean_test_score')] #[yw] the () can be omitted

###prediciton and test error
best_ = grid.best_estimator_
y_test_hat = best_.predict(X_test)
confusion_table(y_test_hat, y_test)
confusion_table(y_test, y_test_hat)

#### Radial Basis kernel
svm_rbf = SVC(kernel="rbf", gamma=1, C=1)
svm_rbf.fit(X_train, y_train)

kfold = skm.KFold(5,
                  random_state=0,
```

```
                  shuffle=True)
grid = skm.GridSearchCV(svm_rbf,
                        {'C':[0.1,1,10,100,1000],
                         'gamma':[0.5,1,2,3,4]},
                        refit=True,
                        cv=kfold,
                        scoring='accuracy');
grid.fit(X_train, y_train)
grid.best_params_
```

9.8.7.2 ROC curve

```
fig, ax = subplots(figsize=(8,8))
roc_curve(best_svm,
          X_train,
          y_train,
          name='Training',
          color='r',
          ax=ax);
```

9.8.7.3 SVM with multiple classes

```
svm_rbf_3 = SVC(kernel="rbf",
                C=10,
                gamma=1,
                decision_function_shape='ovo');
svm_rbf_3.fit(X, y)
fig, ax = subplots(figsize=(8,8))
plot_svm(X,
         y,
         svm_rbf_3,
         scatter_cmap=cm.tab10,
         ax=ax)
```

10 Chapter 10: Deep Learning

- NN was popular in the 1980s. Then took a back seat in the 1990s when SVMs, RF, and Boosting were successful. NN reemerged 2010 as CNN and DL started to garner success. By 2020, become dominant and very successful.
- Part of success due to vast improvements in computing power (GPU computing), more training data sets, and advances in algorithms and software: TensorFlow and PyTorch.
- Three prominent figures: Yann LeCun, Geoffrey Hinton and Yoshua Bengio and their students. Three three scientist received the 2019 ACM Turing Award.

10.1 Single Layer Neural Network

The name *neural network* originally derived from thinking of the hidden units as analogous to neurons in the brain. Consider the NN consisting of input layer, one hidden layer and a single output regression unit. Then the output

$$Y = f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \tag{10.1}$$

$$= \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j) \tag{10.2}$$

where,

- $A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ are the *activations* in the hidden layer: they are simply nonlinear transformation via g of an affine transformation of the input features. Each A_k may be understood as a basis function. The success of NN lies in that A_k are not prescribed, rather are learned from data by learning the coefficients w_{kj}.
- g is a *activate function*. E.g.: sigmoid (for binary output unit), softmax (for multi-class output), linear (for regression) or ReLU (for hidden layers). The activation functions typically in the hidden layers are *nonlinear*, allowing to model complex nonlinearities and interactions. otherwise the model collapses to a linear model.
- For regression, the model is fit by minimizing the RSS loss $\sum_{i=1}^{n} (y_i - f(x_i))^2$. **non-convex**

- For classification , if there are M classes, and the logit output for class m is

$$Z_m = \beta_{m0} + \sum_{\ell=1}^{K} \beta_{m\ell} A_\ell$$

where K is the number of activation nodes in the previous layer. The output activation function encodes the *softmax function*

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{\ell=0}^{M} e^{Z_\ell}}$$

The model is then fit by minimizing the *negative log-liklihood* (or cross-entropy)

$$-\sum_{i=1}^{n} \sum_{m=1}^{M} y_{im} \log(f_m(x_i))$$

where y_{im} is *one-hot* coded.

10.2 Multi-layer NN

In theory, a single hidden layer with a large number of units has the ability to approximate most functions (universal approximator). However, with multi-layers each of smaller size, the computation is reduced and better solution is obtained.

10.3 CNN for Image classificaiton

Clinches its success in CV such as classifying images. The CNN builds up an image in a hierarchical fashion. Edges and shapes are recognized in lower layers and piece together to form more complex shapes, eventually assembling the target image. The hierarchical construction is achieved by *convolution* to discover spatial structure. Convolution allows for parameter sharing and finding common small patterns that occur in different parts of the image (feature translation invariance), typically followed by ReLU, sometimes separately called a *detector layer*) and *pooling* (to summarize, for down-sampling to select a prominent subset and allowing location invariance).

CNN convolves a small filter (image, typically small, e.g., 3×3) representing a small shape, edge, etc. with an input image by sliding the filer around the input image, **scoring the match** by *dot-product*. the more match, the higher the score is. Each filter has the same number of channels as that of the input layer. The filter is typically *learned* by the network via a learning algorithm. The result of the convolution is a new feature map. The convolved image highlights regions of the original image that resemble the convolution filter.

Architecture of a CNN - many convolve-then-pool layers. Sometimes, we repeat several convolve layers before a pool layer. This effectively increases the dimension of the filter. - each filter creates a new channel in the convolution layer. - As pooling reduces the size, the number of filters/channel is typically increased. - network can be very deep. - As pooling has reduced each channel feature map down to a few pixels in each dimension, at the point, the 3D feature maps are *flattened*, and fed into one or more *FC* layers before reaching to the output layey.

10.4 RNN and LSTM

There are many sequence data such as sentence, time series, speech, music etc. RNN build models that take into account the sequential nature of the data and build a memory of the past.

- the feature for each observation is a *sequence* of vectors $X = \{X_1, X_2, \cdots, X_L\}$
- the target Y: a single variable (e.g. binary variable), one-hot vector (multiclass). Can also be a sequence (`seq2seq`), e.g., translation in a different language.
- The hidden layer is a sequence of vectors A_ℓ receiving X_ℓ and $A_{\ell-1}$ as inputs and output O_ℓ. The weight matrices are shared at different time step, hence the name *recurrent*. A_ℓ accumulates a history of what has been seen and represents an evolving model that is updated when X_ℓ is processed.
- Suppose $X_\ell = (X_{\ell 1}, \cdots, X_{\ell p})$, and $A_\ell = (A_{\ell 1}), \cdots, A_{\ell K}$, then $A_{\ell k}$ and O_ℓ are computed by

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1 s}\right)$$

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}$$

  If we are only interested in the predicting O_L at the last unit, then for squared error loss, and n sequence/response pairs (examples), we minimize

$$\sum_{i=1}^{n} (y_i - O_{iL})^2 = \sum_{i=1}^{n} \left(y_i - (\beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{\ell j}^{[i]} + \sum_{s=1}^{K} u_{ks} a_{\ell-1 s}^{[i]}))\right)^2$$

- Deep RNN: having more than one hidden layers in an RNN. The sequence A_ℓ is treated as an input sequence to the next hidden layers.
- in LSTM, two tracks of hidden layer activation are maintained; each A_ℓ receive the short memory $A_{\ell-1}$, as well as from a long memory that reaches further back in time.
- bi-drectional RNN

10.5 Applications

10.5.1 Language models

Application: Sentiment Analysis (document classification)

10.5.1.1 Bag-of-words

How to create features for a document contains a sequence of L words?

- Form a dictionary, e.g. most frequently used 10K words e.g., occuring in the training documents
- create a binary vector of length $p = 10K$ for each document, and score 1 in every position that the corresponding word occurred. (Bag-of-words)
- With n documents, this will create a $n \times p$ *sparse* feature matrix.
- Bag-of-words are *unigrams*, we can also use *bigrams* (occurrences of adjacent word pairs), and in general m-grams$, to take into account the *context*.
- one could also record the relative frequency of words.

10.5.1.2 Word embeddings

- Each document is represented as a sequence of words $\{\mathcal{W}_\ell\}_{\ell=1}^{L}$. Typically we truncate/pad the documents to the same number of L words (e.g. L= 512)
- Each word is represented as a *one-hot* encoded binary vector of X_ℓ of length $10K$, extremely sparse, would not work well.
- Use an embedding layer (either pre-trained (trained on large corpus by such as PCA, such as `word2vec`, or `GloVe`) or learned specifically as part of the optimization) to obtain a lower-dimensional *word embedding* matrix $\mathbf{E}$ ($m \times 10K$) to convert each word's binary feature vector of length 10K to a real feature vector of dimension of length m (e.g. 128, 256, 512, 1024.)

10.5.2 Transfering leraning

By freezing the weights of one or a few top layers of a pretrained NN, one can train a new model by only training the last few layers with much **less** training data, yet obtain a good new model. This is because the feature maps (knowledge) learned in the hidden layer may be transferred to a similar task.

10.5.3 Time Series

- *autocorrelation* at lag ℓ: is the correlation of all pairs $(v_t, v_{t-\ell})$ that are ℓ time interval apart.
- order-L autoregression model ($AR(L)$):

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \cdots + \hat{\beta}_L v_{t-L}.$$

  The model can be fit by OLS.
- Use RNN to model a time series by exacting many short mini-series of the form $X = \{X_1, X_2, \cdots, X_L\}$, and a corresponding target Y.
- time series can also be modelled using 1-D CNN.

10.6 When to use Deep Learning

- CNN big success in CV: e.g. image recolonization
- RNN success in sequence data: e.g.: language translation
- when dataset is large, overfitting is not a problem
- Occam's razor: among algorithms performing equally well, the simpler is preferred as it is easier to interpret.

10.7 Fitting a NN: Gradient Descent

Let the loss be $R(\theta)$, where θ is the parameter to be optimized such that the loss is minimized. The loss R is typically a *non-convex* function of the parameters, hence there might be multiple solutions and many local minima. The gradient method updates the parameter by

$$\theta^{t+1} = \theta_t - \rho \nabla R(\theta^t)$$

where ρ is a *learning rate*, a hyper-parameter, e.g. $\rho = 0.01$; and $\nabla R(\theta^t) = \frac{\partial R(\theta)}{\partial \theta}|_{\theta=\theta^t}$ is the gradient of R. The gradient can be found by the *backproparation* using the *chain rule*. The backpropation distributes a fraction of residual $y_i - f_\theta(x_i)$ at each observation i to each parameter via the hidden units. Modern software such as Tensorflow or PyTorch can easily compute the gradient of a function.

When overfitting is detected, training stops. Since R is non-convex, in general we can hope to end up at a good local minimum.

- the learning rate ρ must be carefully chosen, typically cannot be too large. *Early stopping* (a kind of regularization) may help.

- minibatch: rather than using *all* data each step to update the parameter, draw a random minibatch sample at each step to update the parameter via gradient descent. Such a method is called *SGD*. Minibatch size is a hyperparameter, e.g. 128. It balances bias and variance. It turns out SGD imposes a regularization similar to ridge.
- epoch: One epoch sweeps through the entire training data set with the number of minibatch subsets that is determined by

$$\text{number of minibatches in one epoch} = \frac{n}{\text{minibatch size}}$$

- regularization: lasso, ridge; the hyperparameter λ may vary for different layers.
- dropout: at each SGD update, randomly remove units (by setting their activations zero) with probability ϕ (reduce number of variables hence variance ϕ may vary for different layers), and scale up those retained by $1/(1-\phi)$ to compensate. Dropout has similar effect to ridge.
- data augmentation: make many copies of (x_i, y_i), distort each copy by
  - adding a small amount of noise (e.g. Gaussian) to the x_i,
  - zooming, horizontal and vertical shifting, shearing, small rotation, flipping.

  but leave y_i alone. This effectively has increased the training set. This make the model *robust* to small perturbation in x_i, equivalent to ridge. especially effective with SGD in CV with minibatch where augmented images are added on-the-fly without the need to store them.

10.8 Interpolation and Double Descent

- For a OLS model, When the degree of freedom of a model $d <= n$, the number of examples, we see usual bias-variance trade-off. When $d = n$, it's an interpolating polynomial, very wiggly.
- when $d > n$, the training error is zero, and there are no unique solutions. Among the zero-residual solutions, if pick the *minimum-norm* (hence the smoothest) solution, with the increased dof, it's easy for the model not only fit the training data, but also decreased $\sum_{j=1}^{d} \hat{\beta}_j^2$ (there is no need to have large β_j to fit the training data), leads to solutions actually generalize well with small variance (on test data). An interpolating model may perform better than a slightly less complex model that does not interpolate the data. This phenomenon is called *double descent*.

Such a minimum norm solution may be obtained by SGD with a small learning rate. In this case, the SGD solution path is similar to ridge path. - By analogy, deep and wide NN fit by SGD down to zero training erro often give good solutions that generalize well. - In particular cases with high signal-to-noise-ratio (SNR $= \frac{Var[f(x)]}{\sigma^2}$), where f is the signal and σ^2 is the noise variance (irreducible error). e.g., image recognition, the NN is less prone to overfitting.

- Double descent doesn't contradict the bias-variance trade-off. rather it reveas that the number of basis functions does not properly capture the true model "complexity". In other words, a minimum norm solution with d dof has lower flexibility than the model with d dof.

Most statistical learning method with regularization do not exhibit double descent, as they do not interpolate data, but still achieve good result.

- Maximal margin classifier and SVM that have zero training error often achieve very good test error, this is because they seek smooth minimum norm solutions.

10.9 Homework:

- Conceptual: 1–5
- Applied: At least one.

10.10 Code Snippet

10.10.1 Python

```
#### in Windows the code below is true.
'logs\\hitters\\version_0\\metrics.csv' == r'logs\hitters\version_0\metrics.csv'
del Hitters  # delete the object

[f for f in glob('book_images/*')] # get a list of file names from the dir: book_images

' '.join(lookup[i] for i in sample_review)
```

10.10.2 Numpy

```
X_test.astype(np.float32)
coefs = np.squeeze(coefs)
```

10.10.3 Pandas

```
Y = Hitters['Salary'].to_numpy()

labs = json.load(open('imagenet_class_index.json'))
class_labels = pd.DataFrame([(int(k), v[1]) for k, v in
                             labs.items()],
                             columns=['idx', 'label'])
class_labels = class_labels.set_index('idx')
class_labels = class_labels.sort_index() # sort the rows of a pandas dataframe by index

img_df = img_df.sort_values(by='prob', ascending=False)[:3]
img_df.reset_index().drop(columns=['idx'])

pd.merge(X,
                  pd.get_dummies(NYSE['day_of_week']),
                  on='date')

X = X.reindex(columns=ordered_cols)
```

10.10.4 Graphics

10.10.5 ISLP and statsmodels

10.10.6 sklearn

10.10.6.1 Linear Regression

```
hit_lm = LinearRegression().fit(X_train, Y_train)
Yhat_test = hit_lm.predict(X_test)
np.abs(Yhat_test - Y_test).mean()

M.score(X[~train], Y[~train])  # M is a lm, .score for R^2.
```

10.10.6.2 Lasso

```
scaler = StandardScaler(with_mean=True, with_std=True)
lasso = Lasso(warm_start=True, max_iter=30000)
standard_lasso = Pipeline(steps=[('scaler', scaler),
                                 ('lasso', lasso)])

### Calculate the lambda values
X_s = scaler.fit_transform(X_train)
n = X_s.shape[0]
lam_max = np.fabs(X_s.T.dot(Y_train - Y_train.mean())).max() / n
param_grid = {'alpha': np.exp(np.linspace(0, np.log(0.01), 100))
              * lam_max}


cv = KFold(10,
           shuffle=True,
           random_state=1)
grid = GridSearchCV(lasso,
                    param_grid,
                    cv=cv,
                    scoring='neg_mean_absolute_error')
grid.fit(X_train, Y_train);

trained_lasso = grid.best_estimator_
Yhat_test = trained_lasso.predict(X_test)
np.fabs(Yhat_test - Y_test).mean()
```

10.10.6.3 torch for non-linear regression

```
class HittersModel(nn.Module):

    def __init__(self, input_size): #input_size = feature_dim
        super(HittersModel, self).__init__()
        self.flatten = nn.Flatten()
        self.sequential = nn.Sequential(
            nn.Linear(input_size, 50),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(50, 1))

    def forward(self, x):
```

```
        x = self.flatten(x)
        return torch.flatten(self.sequential(x))

hit_model = HittersModel(X.shape[1])

summary(hit_model,
        input_size=X_train.shape,
        col_names=['input_size',
                   'output_size',
                   'num_params']) # indicate the columns included in the summary
#### Form dataset
X_train_t = torch.tensor(X_train.astype(np.float32))
Y_train_t = torch.tensor(Y_train.astype(np.float32))
hit_train = TensorDataset(X_train_t, Y_train_t)
X_test_t = torch.tensor(X_test.astype(np.float32))
Y_test_t = torch.tensor(Y_test.astype(np.float32))
hit_test = TensorDataset(X_test_t, Y_test_t)

max_num_workers = rec_num_workers()

#### Form data module
hit_dm = SimpleDataModule(hit_train,
                          hit_test, #test dataset
                          batch_size=32,
                          num_workers=min(4, max_num_workers),
                          validation=hit_test)
### setup optimizer, loss function and additional error metrics
hit_module = SimpleModule.regression(hit_model, # using default square loss for training
                           metrics={'mae':MeanAbsoluteError()}) # additional metric
#### set up traning logger
hit_logger = CSVLogger('logs', name='hitters')

#### Training the model
hit_trainer = Trainer(deterministic=False, # deterministic=True is not working when using GPU
                      max_epochs=50,
                      log_every_n_steps=5,
                      logger=hit_logger,
                      callbacks=[ErrorTracker()])
hit_trainer.fit(hit_module, datamodule=hit_dm)

#### Evaluate the test error
hit_trainer.test(hit_module, datamodule=hit_dm)
```

```
### Make prediction
hit_model.eval()
preds = hit_module(X_test_t)
torch.abs(Y_test_t - preds).mean()
```

10.10.6.4 Torch for nonlinear classificaiton

```
### Data Module
mnist_dm = SimpleDataModule(mnist_train,
                            mnist_test,
                            validation=0.2,
                            num_workers=max_num_workers,
                            batch_size=256)
class MNISTModel(nn.Module):
    def __init__(self):
        super(MNISTModel, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 256),
            nn.ReLU(),
            nn.Dropout(0.4))
        self.layer2 = nn.Sequential(
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3))
        self._forward = nn.Sequential(
            self.layer1,
            self.layer2,
            nn.Linear(128, 10))
    def forward(self, x):
        return self._forward(x)

mnist_model = MNISTModel()
summary(mnist_model,
        input_data=X_, #also ok X_.shape or [256, 1, 28, 28]
        col_names=['input_size',
                   'output_size',
                   'num_params'])
### Setup loss, optimizer, additional metrics
mnist_module = SimpleModule.classification(mnist_model,
                                           num_classes=10)
```

```
mnist_logger = CSVLogger('logs', name='MNIST')

### Model training
mnist_trainer = Trainer(deterministic=False,
                        max_epochs=30,
                        logger=mnist_logger,
                        callbacks=[ErrorTracker()])
mnist_trainer.fit(mnist_module,
                  datamodule=mnist_dm)

### Evaluating test error
mnist_trainer.test(mnist_module,
                   datamodule=mnist_dm)
```

10.10.6.5 Using Torch for multi-class Logistic Regression

```
class MNIST_MLR(nn.Module):
    def __init__(self):
        super(MNIST_MLR, self).__init__()
        self.linear = nn.Sequential(nn.Flatten(),
                                    nn.Linear(784, 10))
    def forward(self, x):
        return self.linear(x)

mlr_model = MNIST_MLR()
mlr_module = SimpleModule.classification(mlr_model,
                                         num_classes=10)
mlr_logger = CSVLogger('logs', name='MNIST_MLR')

mlr_trainer = Trainer(deterministic=False,
                      max_epochs=30,
                      callbacks=[ErrorTracker()])
mlr_trainer.fit(mlr_module, datamodule=mnist_dm)
mlr_trainer.test(mlr_module,
                 datamodule=mnist_dm)
```

10.10.6.6 Torch for classificaiton (Binary Sentiment Analsyis)

```
max_num_workers=10
(imdb_train,
```

```
 imdb_test) = load_tensor(root='data/IMDB')
imdb_dm = SimpleDataModule(imdb_train,
                           imdb_test,
                           validation=2000,
                           num_workers=min(6, max_num_workers),
                           batch_size=512)

class IMDBModel(nn.Module):

    def __init__(self, input_size):
        super(IMDBModel, self).__init__()
        self.dense1 = nn.Linear(input_size, 16)
        self.activation = nn.ReLU()
        self.dense2 = nn.Linear(16, 16)
        self.output = nn.Linear(16, 1)

    def forward(self, x):
        val = x
        for _map in [self.dense1,
                     self.activation,
                     self.dense2,
                     self.activation,
                     self.output]:
            val = _map(val)
        return torch.flatten(val)

imdb_model = IMDBModel(imdb_test.tensors[0].size()[1])
summary(imdb_model,
        input_size=imdb_test.tensors[0].size(),
        col_names=['input_size',
                   'output_size',
                   'num_params'])


imdb_optimizer = RMSprop(imdb_model.parameters(), lr=0.001)
imdb_module = SimpleModule.binary_classification(
                         imdb_model,
                         optimizer=imdb_optimizer)

imdb_logger = CSVLogger('logs', name='IMDB')
imdb_trainer = Trainer(deterministic=False,
                       max_epochs=30,
                       logger=imdb_logger,
```

```
                      callbacks=[ErrorTracker()])
imdb_trainer.fit(imdb_module,
                 datamodule=imdb_dm)

test_results = imdb_trainer.test(imdb_module, datamodule=imdb_dm)
```

10.10.6.7 Torch with CNN

```
cifar_dm = SimpleDataModule(cifar_train, #torch TensorDataSet
                            cifar_test,
                            validation=0.2,
                            num_workers=max_num_workers,
                            batch_size=128)

class BuildingBlock(nn.Module):

    def __init__(self,
                 in_channels,
                 out_channels):

        super(BuildingBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels=in_channels,
                              out_channels=out_channels,
                              kernel_size=(3,3),
                              padding='same')
        self.activation = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=(2,2))

    def forward(self, x):
        return self.pool(self.activation(self.conv(x)))

class CIFARModel(nn.Module):

    def __init__(self):
        super(CIFARModel, self).__init__()
        sizes = [(3,32),
                 (32,64),
                 (64,128),
                 (128,256)]
        self.conv = nn.Sequential(*[BuildingBlock(in_, out_)
                                    for in_, out_ in sizes])
```

```
        self.output = nn.Sequential(nn.Dropout(0.5),
                                    nn.Linear(2*2*256, 512),
                                    nn.ReLU(),
                                    nn.Linear(512, 100))
    def forward(self, x):
        val = self.conv(x)
        val = torch.flatten(val, start_dim=1) # flatten starting from dim=1 (default), the fi
        return self.output(val)

cifar_model = CIFARModel()
summary(cifar_model,
        input_data=X_,
        col_names=['input_size',
                   'output_size',
                   'num_params'])

### define loss, optimizer, etc
cifar_optimizer = RMSprop(cifar_model.parameters(), lr=0.001)
cifar_module = SimpleModule.classification(cifar_model,
                                    num_classes=100,
                                    optimizer=cifar_optimizer)
cifar_logger = CSVLogger('logs', name='CIFAR100')

# Training
cifar_trainer = Trainer(deterministic=False,
                        max_epochs=30,
                        logger=cifar_logger,
                        callbacks=[ErrorTracker()])
cifar_trainer.fit(cifar_module,
                  datamodule=cifar_dm)

cifar_trainer.test(cifar_module,
                   datamodule=cifar_dm)
```

10.10.6.8 Transfer learning

```
### Pre-processing
resize = Resize((232,232), antialias=True) #target size: (232,232)
crop = CenterCrop(224) #centered and cropped to target size 224
normalize = Normalize([0.485,0.456,0.406],   # mean for each channel
                      [0.229,0.224,0.225])   # std for each channel
```

```
imgfiles = sorted([f for f in glob('book_images/*')])
imgs = torch.stack([torch.div(crop(resize(read_image(f))), 255) # element-wise div by 255
                    for f in imgfiles])
imgs = normalize(imgs)

resnet_model = resnet50(weights=ResNet50_Weights.DEFAULT) # get the model
resnet_model.eval()
img_preds = resnet_model(imgs) #logit values

img_probs = np.exp(np.asarray(img_preds.detach())) # convert to propbabilities
img_probs /= img_probs.sum(1)[:,None] # the sum is along the row.
```

10.10.6.9 RNN/LSTM with Torch for Documenation Classification

```
imdb_seq_dm = SimpleDataModule(imdb_seq_train,
                               imdb_seq_test,
                               validation=2000,
                               batch_size=300,
                               num_workers=min(6, max_num_workers)
                               )


class LSTMModel(nn.Module):
    def __init__(self, input_size):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(input_size, 32)
        self.lstm = nn.LSTM(input_size=32,
                            hidden_size=32,
                            batch_first=True)
        self.dense = nn.Linear(32, 1)
    def forward(self, x):
        val, (h_n, c_n) = self.lstm(self.embedding(x))
        return torch.flatten(self.dense(val[:,-1])) # select the last time step of val

lstm_model = LSTMModel(X_test.shape[-1])
summary(lstm_model,
        input_data=imdb_seq_train.tensors[0][:10],
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

```
lstm_module = SimpleModule.binary_classification(lstm_model)
lstm_logger = CSVLogger('logs', name='IMDB_LSTM')

lstm_trainer = Trainer(deterministic=False,
                       max_epochs=20,
                       logger=lstm_logger,
                       callbacks=[ErrorTracker()])
lstm_trainer.fit(lstm_module,
                 datamodule=imdb_seq_dm)
lstm_trainer.test(lstm_module, datamodule=imdb_seq_dm)
```

10.10.6.10 RNN/LSTM with Torch for Time Seires Prediciton

```
class NYSEModel(nn.Module):
    def __init__(self):
        super(NYSEModel, self).__init__()
        self.rnn = nn.RNN(3, # number of features
                          12,
                          batch_first=True)
        self.dense = nn.Linear(12, 1)
        self.dropout = nn.Dropout(0.1)
    def forward(self, x):
        val, h_n = self.rnn(x)
        val = self.dense(self.dropout(val[:,-1]))
        return torch.flatten(val)
nyse_model = NYSEModel()

datasets = []
for mask in [train, ~train]:
    X_rnn_t = torch.tensor(X_rnn[mask].astype(np.float32))
    Y_t = torch.tensor(Y[mask].astype(np.float32))
    datasets.append(TensorDataset(X_rnn_t, Y_t))
nyse_train, nyse_test = datasets

nyse_dm = SimpleDataModule(nyse_train,
                           nyse_test,
                           num_workers=min(4, max_num_workers),
                           validation=nyse_test,
                           batch_size=64)

nyse_optimizer = RMSprop(nyse_model.parameters(),
                         lr=0.001)
```

```
nyse_module = SimpleModule.regression(nyse_model,
                                      optimizer=nyse_optimizer,
                                      metrics={'r2':R2Score()})

nyse_trainer = Trainer(deterministic=False,
                       max_epochs=200,
                       callbacks=[ErrorTracker()])
nyse_trainer.fit(nyse_module,
                 datamodule=nyse_dm)
nyse_trainer.test(nyse_module,
                  datamodule=nyse_dm)
```

10.10.6.11 Linear and Nonlinear AR with Torch

```
### Nonlinear-AR model
day_dm = SimpleDataModule(day_train,
                          day_test,
                          num_workers=min(4, max_num_workers),
                          validation=day_test,
                          batch_size=64)

class NonLinearARModel(nn.Module):
def __init__(self):
    super(NonLinearARModel, self).__init__()
    self._forward = nn.Sequential(nn.Flatten(), #flatten a multi-dim tensor into a 1-d tenso
    nn.Linear(20, 32),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(32, 1))
def forward(self, x):
    return torch.flatten(self._forward(x))

nl_model = NonLinearARModel()
nl_optimizer = RMSprop(nl_model.parameters(),
                           lr=0.001)
nl_module = SimpleModule.regression(nl_model,
                                      optimizer=nl_optimizer,
                                      metrics={'r2':R2Score()})

nl_trainer = Trainer(deterministic=False,
                         max_epochs=20,
                         callbacks=[ErrorTracker()])
```

```
nl_trainer.fit(nl_module, datamodule=day_dm)
nl_trainer.test(nl_module, datamodule=day_dm)
```

10.10.7 Useful code snippets

10.10.7.1 Plotting training/validation learning curve

```
def summary_plot(results,
                 ax,
                 col='loss',
                 valid_legend='Validation',
                 training_legend='Training',
                 ylabel='Loss',
                 fontsize=20):
    for (column,
         color,
         label) in zip([f'train_{col}_epoch',
                        f'valid_{col}'],
                       ['black',
                        'red'],
                       [training_legend,
                        valid_legend]):
        results.plot(x='epoch',
                     y=column,
                     label=label,
                     marker='o',
                     color=color,
                     ax=ax)
    ax.set_xlabel('Epoch')
    ax.set_ylabel(ylabel)
    return ax

fig, ax = subplots(1, 1, figsize=(6, 6))
ax = summary_plot(hit_results,
                  ax,
                  col='mae',
                  ylabel='MAE',
                  valid_legend='Validation (=Test)')
ax.set_ylim([0, 400])
ax.set_xticks(np.linspace(0, 50, 11).astype(int));
```

10.10.7.2 Viewing a set of images

```
fig, axes = subplots(5, 5, figsize=(10,10))
rng = np.random.default_rng(4)
indices = rng.choice(np.arange(len(cifar_train)), 25,
                     replace=False).reshape((5,5))
for i in range(5):
    for j in range(5):
        idx = indices[i,j]
        axes[i,j].imshow(np.transpose(cifar_train[idx][0],
                                      [1,2,0]), # transpose the channel to the last dim for c
                                      interpolation=None)
        axes[i,j].set_xticks([])
        axes[i,j].set_yticks([])
```

10.10.7.3 using sk-learn LogisticRegression() with Lasso

```
#### Defining the \lambda
lam_max = np.abs(X_train.T * (Y_train - Y_train.mean())).max() # this is not divided by n, di
lam_val = lam_max * np.exp(np.linspace(np.log(1),
                                       np.log(1e-4), 50))

logit = LogisticRegression(penalty='l1',
                           C=1/lam_max,
                           solver='liblinear',
                           warm_start=True,
                           fit_intercept=True)

coefs = []
intercepts = []

for l in lam_val:
    logit.C = 1/l
    logit.fit(X_train, Y_train)
    coefs.append(logit.coef_.copy())
    intercepts.append(logit.intercept_)
```

10.10.7.4 Viewing the Lasso results with lambda values

```
fig, axes = subplots(1, 2, figsize=(16, 8), sharey=True)
```

```
for ((X_, Y_),
     data_,
     color) in zip([(X_train, Y_train),
                    (X_valid, Y_valid),
                    (X_test, Y_test)],
                    ['Training', 'Validation', 'Test'],
                    ['black', 'red', 'blue']):
    linpred_ = X_ * coefs.T + intercepts[None,:]
    label_ = np.array(linpred_ > 0)
    accuracy_ = np.array([np.mean(Y_ == l) for l in label_.T])
    axes[0].plot(-np.log(lam_val / X_train.shape[0]), #lambda is rescaled by diving N for onl
                 accuracy_,
                 '.--',
                 color=color,
                 markersize=13,
                 linewidth=2,
                 label=data_)
axes[0].legend()
axes[0].set_xlabel(r'$-\log(\lambda)$', fontsize=20)
axes[0].set_ylabel('Accuracy', fontsize=20)
```

10.10.7.5 Insert lags to a time series

```
for lag in range(1, 6):
    for col in cols:
        newcol = np.zeros(X.shape[0]) * np.nan
        newcol[lag:] = X[col].values[:-lag]
        X.insert(len(X.columns), "{0}_{1}".format(col, lag), newcol)#insert at the end
X.insert(len(X.columns), 'train', NYSE['train']) #insert the col training identifier

X = X.dropna() # drop rows with nan
```

10.10.7.6 Preparing time series data for RNN in Torch

```
Y, train = X['log_volume'], X['train']
X = X.drop(columns=['train'] + cols)

ordered_cols = []
for lag in range(5,0,-1):
    for col in cols:
        ordered_cols.append('{0}_{1}'.format(col, lag))
```

```
X = X.reindex(columns=ordered_cols)

X_rnn = X.to_numpy().reshape((-1,5,3))
```

11 Chapter 11: Survival Analysis

Survival Analysis: similar to linear regression, but the outcome variable: *time until an event occurs* (event time, also called survival time, denoted by T), is **censored** at the time (*censoring time* C) when data is collected: for some x_i (e.g. patient health data), the time to event (such as death) is not available at the time of censoring C (e.g. five years after the study began), because the event has not occurred (e.g. patients drop out of the study, patients still alive). Ordinary regression solution would discard such a data (of surviving patients) with x_i, but Survival Analysis do not, because the fact they survived after the censoring time (e.g. five years) is valuable.

- Applications: medicine study for predicting the survival time of patients after receiving a treatment
- customer churn: (predict) the event when customers cancel subscription to a service

11.1 Survial and Censoring Time

- We observe either T or C, i.e., we observe the r.v.

$$Y = \min(T, C)$$

- Define the *status indicator*

$$\delta = \begin{cases} 1 & \text{if } T \leq C \\ 0 & \text{if } T > C \end{cases}$$

- Data set format: n pairs $(y_1, \delta_1), \cdots, (y_n, \delta_n)$.
- *independent censoring*: In general, we assume that conditional on the features, T is *independent* of C.
- In some situation, the above assumption is false. For example, When a patient drop out a study because the patient is too sick. In this case obtained C may lead to overestimate of T.
- *right censoring*: when $Y \leq T$.
- *left censoring*: when $T \leq Y$.
- *interval censoring*: The exact T is not know, but we know it falls in some interval.

11.2 The Survival Curve

- Survival function
$$S(t) = Pr(T > t)$$
is a *decreasing function* that quantifies the probability of surviving past time t. For example, if T represents the time when a customer churns, then $S(t)$ is the probability a customer cancels later than t. The larger $S(t)$ is, the less likely that the customer will cancel before time t.

11.2.1 How to estimate $S(t)$

11.2.1.1 Kaplan-Meier Survival Curve

Let $d_1 < d_2 < \cdots < d_K$ be the unique death (event) times among the non-censored patients, r_k is the number of patients (*risk set*) at risk (still alive) at d_k, and q_k is the number of patients who died at d_k.

The idea is to use *sequential construction*. Use the total probability,

$$Pr(T > d_k) = Pr(T > d_k | T > d_{k-1}) Pr(T > d_{k-1}) + Pr(T > d_k | T \leq d_{k-1}) Pr(T \leq d_{k-1})$$

Note $Pr(T > d_k | T \leq d_{k-1}) = 0$. Therefore

$$S(d_k) = Pr(T > d_k | T > d_{k-1}) S(d_{k-1}) = Pr(T > d_k | T > d_{k-1}) \times \cdots \times Pr(T > d_2 | T > d_1) Pr(T > d_1)$$

It is natural to estimate
$$\hat{Pr}(T > d_j | T > d_{j-1}) = (r_j - q_j)/r_j$$
This leads to the *Kaplan-Meier* estimator

$$\hat{S}(d_k) = \prod_{j=1}^{k} \left(\frac{r_j - q_j}{r_j} \right)$$

for $d_j < d_{k+1}$, set $\hat{S}(t) = \hat{S}(d_k)$, this leads to a step function.

11.2.1.2 the log-rank test to compare the survial curves of two groups

Further, let r_{ik} be the number of patients who are at risk at d_k and q_{ik} be the number of patients who died at d_k, for group i, $i = 1, 2$.

$$r_{1k} + r_{2k} = r_k \qquad q_{1k} + q_{2k} = q_k$$

We can construct the following 2X2 table at each d_k:

	Group 1	Group 2	Total
Died	q_{1k}	q_{2k}	q_k
Survival	$r_{1k} - q_{1k}$	$r_{2k} - q_{2k}$	$r_k - q_k$
Total	r_{1k}	r_{2k}	r_k

To test $H_0 : E(X) = \mu$ for a rv X, construct the statistic

$$W = \frac{X - E(X)}{\sqrt{Var(X)}}$$

where $E(X)$ and $Var(X)$ are under H_0. Let $X = \sum_{k=1}^{K} q_{1k}$, the total death of group 1 up to time d_K. Assume that q_{1k} are uncorrelated. Plug in the formula for X, one can obtain

$$W = \frac{\sum_{k=1}^{K}(q_{1k} - E(q_{1k}))}{\sqrt{\sum_{k=1}^{K} Var(q_{1k})}} = \frac{\sum_{k=1}^{K}(q_{1k} - \frac{q_k}{r_k} r_{1k})}{\sqrt{\sum_{k=1}^{K} \frac{q_k (r_{1k}/r_k)(1 - r_{1k}/r_k)(r_k - q_k)}{r_k - 1}}}$$

When the sample size is large, W has approximately a standard normal distribution. The null Hypothesis for the log-rank test is that there is no difference between the survival curves in the two groups.

11.3 Regression models with a survival response

The goal is to predict the survival time T. The observation is (Y, δ). However fit a regression of $\log Y$ with $Y = \min(T, C)$ on X is not sound because of the censoring. To overcome this difficulty, use the idea of *sequential construction* again as in Kaplan-Meier survival curve. To this end, define the *hazard function* (also called *hazard rate*, or *force of mortality*) as

$$h(t) = \lim_{\Delta t \to 0} \frac{Pr(t < T \leq t + \Delta t | T > t)}{\Delta t}$$

where T is the (true) survival time. The hazard rate is the death rate (in fact the pdf for T conditional on $T > t$) in the instant after t, given survival up to that time. By definition of h, we can derive

$$h(t) = \frac{f(t)}{S(t)}$$

where

$$f(t) = \lim_{\Delta t \to 0} \frac{Pr(t < T \leq t + \Delta t)}{\Delta t}$$

is the *pdf* associated with T. Specifically,

$$f(t) = \frac{d}{dt} F(t) = \frac{d}{dt}[1 - S(t)] = \frac{d}{dt}[1 - \int_0^t h(u)du]$$

The likelihood associated with the i-th observation is [

$$L_i = \begin{cases} f(y_i) & \text{if the } i\text{-th observation is not censored, } \delta_i = 1 \\ S(y_i) & \text{if the } i\text{-th observation is censored, } \delta_i = 0 \end{cases} \tag{11.1}$$

$$= f(y_i)^{\delta_i} S(y_i)^{1-\delta_i} \tag{11.2}$$

] Assume the n observations are independent, the likelihood for the data is

$$L = \prod_{i=1}^{n} f(y_i)^{\delta_i} S(y_i)^{1-\delta_i} = \prod_{i=1}^{n} h(y_i)^{\delta_i} S(y_i).$$

Some reasonable assumptions for f is:

- Exponential: $f(t) = \lambda \exp(-\lambda t)$
- Gamma or Weibull family
- non-parametrically such as by Kaplan-Meirer estimator.

With $h(t)$, define the Cox's *proportional hazard* for an individual with feature vector $x_i = (x_{i1}, \cdots, x_{ip})$ by

$$h(t|x_i) = h_0(t) \exp\left(\sum_{j=1}^{p} x_{ij}\beta_j\right)$$

where, $h_0(t) \geq 0$ is the *baseline hazard* function for an individual with features $x_{i1} = \cdots = x_{ip} = 0$. h_0 can take any functional form. The term $\sum_{j=1}^{p} xij\beta_j$ is called the *relative risk*. The model assumes that one-unit increase in x_{ij} corresponds to an increase in $h(t|x_i)$ by a factor of $\exp(\beta_j)$.

- There is *no intercept* in the model, as it can be absorbed into $h_0(t)$
- The model can easily handle time-dependent covariate: simply replace x_{ij} with $x_{ij}(t)$ for the j-th covariate of i-th observation .
- To check the proportional hazards assumption: for qualitative feature, plot the log hazard function for each level of the feature, the log hazard functions should differ by a constant. For quantitative feature, we can take a similar approach by stratifying the feature.

We cannot directly estate β_j by maximum likelihood using $h(t|x_i)$ because h_0 is not known. But we can use *partial likelihood*. Use the same "sequential in time" logic, the probability that the i-th observation fails at time y_i is:

$$\frac{h_0(y_i) \exp\left(\sum_{j=1}^{p} x_{ij}\beta_j\right)}{\sum_{i':y_{i'} \geq y_i} h_0(y_i) \exp\left(\sum_{j=1}^{p} x_{i'j}\beta_j\right)} = \frac{\exp\left(\sum_{j=1}^{p} x_{ij}\beta_j\right)}{\sum_{i':y_{i'} \geq y_i} \exp\left(\sum_{j=1}^{p} x_{i'j}\beta_j\right)}$$

which is called *relative risk function* at y_i.

The *partial likelihood* (approximation to the likelihood) over all of the uncensored observations is

$$PL(\beta) = \prod_{i:\delta_i=1} \frac{\exp\left(\sum_{j=1}^{p} x_{ij}\beta_j\right)}{\sum_{i':y_{i'}\geq y_i} \exp\left(\sum_{j=1}^{p} x_{i'j}\beta_j\right)} = \prod_{i:\delta_i=1} RR_i(\beta)$$

In the above formula, it is assumed there are *no tied failure times*. If there are, then the formula needs to be modified. To estimate β, simply maximize $PL(\beta)$. To estimate β, simply maximize $PL(\beta)$. Lasso or ridge penalty terms of β can be added to obtain shrinkage estimate.

11.3.1 Connection with the log-rank test

- for the case of a single binary covariate, the *score test* for $H_0 : \beta = 0$ in Cox's hazards model is *exactly equal to the log-rank test*.

11.3.2 AUC for Survival Analysis: the C-index

- for each observation, calculate the estimated risk score:

$$\hat{\eta}_i = \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip}, \quad i = 1, \cdots, n$$

- compute Harrell's concordance index (C-index):

$$C = \frac{\sum_{i,i':y_i>y_{i'}} I(\hat{\eta}_{i'} > \hat{\eta}_i)\delta_{i'}}{\sum_{i,i':y_i>y_{i'}} \delta_{i'}}$$

  The C-index is the proportion of pairs for which the model correctly predicts the relative survival time, among all pairs for which this can be determined. E.g., $C = 0.733$ indicates given two random observations from the test set, the model can predict with 73.3% accuracy which will survive.

11.4 Homework:

- Conceptual: 1–
- Applied: At least one.

11.5 Code Snippet

11.5.1 Python

```
rng = np.random.default_rng(10)
N = 2000
Operators = rng.choice(np.arange(5, 16),
                       N,
                       replace=True)

np.clip(W, 0, 1000)
D['Failed'] = rng.choice([1, 0],
                         N,
                         p=[0.9, 0.1])
```

11.5.2 Numpy

11.5.3 Pandas

```
BrainCancer['sex'].value_counts()

#### return mean or mode of columns of a df
def representative(series):
    if hasattr(series.dtype, 'categories'): # hasattr(object, attribute)
        return pd.Series.mode(series)
    else:
        return series.mean()
modal_data = cleaned.apply(representative, axis=0)
```

11.5.4 Graphics

11.5.5 ISLP and statsmodels

```
from lifelines import \
     (KaplanMeierFitter,
      CoxPHFitter)
from lifelines.statistics import \
```

```
     (logrank_test,
      multivariate_logrank_test)
from ISLP.survival import sim_time

### Kaplan-Meier estimator
fig, ax = subplots(figsize=(8,8))
km = KaplanMeierFitter()
km_brain = km.fit(BrainCancer['time'], BrainCancer['status'])
km_brain.plot(label='Kaplan Meier estimate', ax=ax)
```

11.5.5.1 stratified K-M estimator

```
fig, ax = subplots(figsize=(8,8))
by_sex = {}
for sex, df in BrainCancer.groupby('sex'):
    by_sex[sex] = df
    km_sex = km.fit(df['time'], df['status'])
    km_sex.plot(label='Sex=%s' % sex, ax=ax)
```

11.5.5.2 Log-rank test

```
logrank_test(by_sex['Male']['time'],
             by_sex['Female']['time'],
             by_sex['Male']['status'],
             by_sex['Female']['status'])
```

11.5.5.3 Cox proportional Hazards model

```
coxph = CoxPHFitter # shorthand
sex_df = BrainCancer[['time', 'status', 'sex']]
model_df = MS(['time', 'status', 'sex'],
              intercept=False).fit_transform(sex_df) #MS has coded 'sex` column to binary. Ma
cox_fit = coxph().fit(model_df,
                      'time',
                      'status')
cox_fit.summary[['coef', 'se(coef)', 'p']]

cox_fit.log_likelihood_ratio_test()

# fit all variables
all_MS = MS(cleaned.columns, intercept=False)
```

```
all_df = all_MS.fit_transform(cleaned)
fit_all = coxph().fit(all_df,
                      'time',
                      'status')
fit_all.summary[['coef', 'se(coef)', 'p']]
modal_X = all_MS.transform(modal_df)
predicted_survival = fit_all.predict_survival_function(modal_X)
fig, ax = subplots(figsize=(8, 8))
predicted_survival.plot(ax=ax);
```

11.5.6 sklearn

11.5.7 Useful code snippets

12 Class Project

Goal

to use various ML algorithms to predict a meaningful target Y by *classification algorithms* or *regression algorithms*. Present it at COS Research Sympsium in the end of April.

Data set: real-world stock price and volume data. We could start with just one stock, e.g. APAL, S&P 500, index, DJ index.

Some ideas:

- Create one model for each stock.
- create a single model for all stocks. Needs to embed each stock in a feature space. Research?

Start-up code: refer to the page https://github.com/ywanglab/Predicting_stock_movement/blob/main/Time_series_stock_data_analysis_ver2.ipynb. Perform some EDA to feel the data.

reference ticker symbols: https://gist.github.com/quantra-go-algo/ac5180bf164a7894f70969fa563627b2

Questions: Which are the X variables? price, volume, return, day of week, month of year, etc. What is the Y variable? next-day price, next-day return, next-five-day average price, next-five-day-average return, etc.

Models

- Linear regression:
  - including continuous variables (price, volume), categorical variables (day-of-week, month-of-year)
  - transforming X (for including non-linear relation between Y and X) or Y (when Y is heteroschedatic, i.e., with varied ϵ_i)
  - plot residual plot to see if $Var(\epsilon_i)$ is changing. If yes, may appeal to transforming Y, e.g., $\log Y$, $\sqrt{Y}$.
  - investigate outliers (points with unusual Y-values) using the *residual plot* or looking at *studentized residual*.
  - investigate high leverage points (with unusual x values), by calculating *leverage statistics*.
  - Investigate if there is *colinearity* among the variables by calculating *VIF*.

- classification: predicting directions of the stock price movement. binary (with two direction), or multinomial (more than two values: e.g., up, same, down), LDA, QDA
- regularization of the parameters: lasso (L^1), ridge (L^2)
- selection of variables: forward, backward, mixture, regularization, cross-validation
- decision tree: random forest, boosting
- SVM: support vector machines
- NN

13 Summary

In summary, this book has no content whatsoever.

References

James et al. (2023)

James, G., D. Witten, T. Hastie, R. Tibshirani, and J. Taylor. 2023. *An Introduction to Statistical Learning*. USA: Springer. https://hastie.su.domains/ISLP/ISLP_website.pdf.