Initialization

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

In [2]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt

In [3]: from sklearn.compose import ColumnTransformer
   from sklearn.pipeline import Pipeline
   from sklearn.impute import SimpleImputer
   from sklearn.preprocessing import OneHotEncoder
   from sklearn.preprocessing import StandardScaler
   from sklearn.preprocessing import add_dummy_feature
   from sklearn.linear_model import LinearRegression
   from mpl_toolkits.mplot3d import Axes3D
Linear Equations

- From school, the equation of a straight line:
  \[ y = a + bx \]
  E.g. \( y = 3 + 2x \)
- From the point of view of plotting this line, what's \( a \)? What's \( b \)?
- In general
  \[ y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \]
  - \( \beta_0, \ldots, \beta_n \) are numbers, called the **coefficients**
  - \( x_1, \ldots, x_n \) are the variables
  - each of the things being added together is called a **term**
  So a linear equation is the sum of a number of terms, where each term is either a constant or the product of a constant and a variable
- Given a linear equation and the values of the variables \((x_1, \ldots, x_n)\), we can **evaluate** the equation, i.e. work out the value of \( y \)

Class exercises

- Which of these are linear equations?
  1. \( y = 6 + 2x_1 + 4x_3 + x_7 \)
  2. \( y = 6x_1 - 3x_2 \)
  3. \( y = 3 + \sin(x_1) \)
  4. \( y = 3x_0^0 + 7x_1^1 + 19x_3^2 \)
  5. \( y = 3 + 14x_1 x_2 + 12x_3 \)
- Evaluate \( y = 2 + 3x_1 + 4x_2 + 5x_3 \):
  1. in the case that \( x_1 = 1, x_2 = 1, x_3 = 1 \)
  2. in the case that \( x_1 = 0, x_2 = 1, x_3 = 5 \)
Linear Equations and Vectors

- Give a linear equation \( y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \),
  - we can gather the variables into a row vector \([x_1, x_2, \ldots, x_n]\)
  - we can gather the coefficients (except \(\beta_0\)) into a column vector

\[
\begin{bmatrix}
\beta_1 \\
\beta_2 \\
\vdots \\
\beta_n
\end{bmatrix}
\] (of the same dimension, \(n\))

- E.g. from \( y = 12 + 3x_1 + 4x_2 + 5x_3 \), we get \( x = [x_1, x_2, x_3] \) and \( \beta = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \)
- What are the two vectors for \( y = 7 + 20x_1 + x_3 \)?
- Hence, the linear equation \( y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \) can equivalently be written in this form:

\[
y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i
\]
- It can also, equivalently, be written in this form:

\[
y = \beta_0 + x\beta
\]
- Hence, to evaluate a linear equation, simply multiply the two vectors and add \(\beta_0\)

Evaluating a linear equation in numpy

- If you had to evaluate a linear equation, you might be tempted to write a loop:

```python
In [4]: # Evaluate y = 12 + 3x1 + 4x2 + 5x3 in the case where x1=7, x2=3, x3=20
   : y = 12
   : for (beta_i, x_i) in zip(np.array([3, 4, 5]), np.array([7, 3, 20])):
   :     y += beta_i * x_i
   :
Out[4]: 145
```
- But you don't need to write your own loop: use numpy library's matrix multiplication method

```python
In [5]: y = 12 + np.array([3, 4, 5]).dot(np.array([7, 3, 20]))
   : y
Out[5]: 145
```
Linear Equations and Vectors: Tidying the maths

- Give a linear equation $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$,
  - we can gather the variables into a row vector but include an extra variable $x_0$, whose value will always be 1:
    $$[1, x_1, x_2, \ldots, x_n]$$
  - we can gather all the coefficients (including $\beta_0$) into a column vector
    $$\begin{bmatrix}
    \beta_0 \\
    \beta_1 \\
    \beta_2 \\
    \vdots \\
    \beta_n
    \end{bmatrix}$$
    (of the same dimension, $n + 1$)

- E.g. from $y = 12 + 3x_1 + 4x_2 + 5x_3$, we get $x = [1, x_1, x_2, x_3]$ and $\beta = \begin{bmatrix} 12 \\ 3 \\ 4 \\ 5 \end{bmatrix}$

- What are the two vectors for $y = 7 + 20x_1 + x_3$?
- Hence, the linear equation $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$ can equivalently be written in this form:
  $$y = \sum_{i=0}^{n} \beta_i x_i$$

- It can also, equivalently, be written in this form:
  $$y = x\beta$$

- Hence, to evaluate a linear equation, simply multiply the two vectors

```python
In [6]: y = np.array([12, 3, 4, 5]).dot(np.array([1, 7, 3, 20]))
   y
Out[6]: 145
```
Evaluating Linear Equations and Matrices

- Suppose you need to evaluate the same linear equation lots of times — with different values for \( x \):
  - E.g. evaluate \( y = 12 + 3x_1 + 4x_2 + 5x_3 \) for:
    - \( x_1 = 7, x_2 = 3, x_3 = 20 \) and
    - \( x_1 = 10, x_2 = 20, x_3 = 0 \) and
    - \( x_1 = 1, x_2 = 1, x_3 = 1 \) and
    - \( x_1 = 100, x_2 = 0, x_3 = -2 \)
- If we gather the values for the variables into a matrix, \( X \), but with an extra element \( x_i^{(0)} \) in each row, all of which will be 1, then we can obtain all the results by simple matrix multiplication:
  \[
  y = X \beta
  \]
  
  E.g.
  \[
  y = \begin{bmatrix}
  1 & 7 & 3 & 20 \\
  1 & 10 & 20 & 0 \\
  1 & 1 & 1 & 1 \\
  1 & 100 & 0 & -2
  \end{bmatrix}
  \begin{bmatrix}
  12 \\
  3 \\
  4 \\
  5
  \end{bmatrix}
  \]
  
  It produces a vector of results, e.g. \( y = \begin{bmatrix} 145 \\ 122 \\ 24 \\ 302 \end{bmatrix} \)

Evaluating a linear equation multiple times in numpy

- Same story: no loop, use matrix multiplication

```
In [7]: y = np.array([[1, 7, 3, 20], [1, 10, 20, 0], [1, 1, 1, 1], [1, 100, 0, -2]]).dot(np.array([12, 3, 4, 5]))
Out[7]: array([145, 122, 24, 302])
```

- This is **vectorization** again: concise, fast code!

Linear Models

- Recall: We want to learn a model from a labeled training set
- For the remainder of CS4618, we will content ourselves with learning a linear model
  - In regression, we'll try to find a linear equation that best fits the training examples
  - In classification, we'll try to find a linear equation that best separates training examples from different classes
- We'll start with regression and we'll begin by assuming there's only one feature
Linear Regression: with one feature

- We'll read in the (cleaned-up version of the) Cork Property Prices dataset and ignore all features other than `flarea`
- For the purposes of this explanation, we won't scale the data: so no need for a ColumnTransformer
- We'll also extract the prices (the target values)
- Also for the purposes of this explanation, we will use the entire dataset as our training set
  - We will learn later that using all the data for training is usually not the right thing to do

```
In [8]: # Use pandas to read the CSV file
df = pd.read_csv("datasets/dataset_corkA.csv")

# Get the feature-values (just flarea) and the target values
flareas = df["flarea"]
prices = df["price"]
```

```
In [9]: # Plot the data
fig = plt.figure()
plt.title("Training set")
plt.scatter(flareas, prices, color = 'green')
plt.xlabel("Floor area (sq metres)")
plt.xlim(0, 500)
plt.ylabel("Price (000 euros")
plt.ylim(0, 1000)
plt.show()
```

- The goal of our learning algorithm is to fit a linear model to this data:
  \[ \hat{y} = \beta_0 + \beta_1 \times \text{flarea} \]
- In other words, our goal is to choose values for \( \beta_0 \) and \( \beta_1 \)
  - From the point of view of plotting this line, what's \( \beta_0 \)? What's \( \beta_1 \)?
  - E.g. we could choose \( \beta_0 = 800 \) and \( \beta_1 = -5 \)
  - Or we could choose \( \beta_0 = 200 \) and \( \beta_1 = 5 \)
- Let's refer to any particular choice as \( h_\beta \) (h for hypothesis)
  - The first example above is \( h_{[800,-5]} \)
  - The second example above is \( h_{[200,5]} \)
- But there is an infinite set of linear models the algorithm can choose from
  - An infinite number of straight lines it can draw
  - Or, equivalently, an infinite set of values from which it can pick \( \beta_0 \) and \( \beta_1 \)
- We want it to choose the one that best fits the data
Loss functions

- The algorithm needs a function that measures how well a model (hypothesis) fits the data
  - This is called its **loss function**, designated \( J \)
  - The function takes in a particular \( h_\beta \) and gives it a score
    - Low numbers are better!
  - For each \( x \) in the training set, it will compare \( h_\beta(x) \), which is the prediction that \( h_\beta \) makes on \( x \), with the actual value \( y \)
- The loss function most usually used for linear regression is the **mean squared error**: 
  \[
  J(X, y, \beta) = \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2
  \]
  - Why do you think we square the differences? (Two reasons)
    - The best model is the one that **minimizes** the loss function
    - Hence, this is often referred to as **ordinary least-squares regression** (OLS)
  - In fact, we often divide by 2:
    \[
    J(X, y, \beta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2
    \]
    — the ‘winner’ is still the same, but this makes the calculus ‘tidier’ later

The loss function in numpy

- Looks like a loop: work out \( h_\beta \) for each \( x^{(i)} \)
  - But \( h_\beta \) is a linear equation, and we want to evaluate it lots of times (for each example \( x^{(i)} \))
  - So we use the vectorized approach from above (assuming all the examples contain an extra element, \( x_0^{(i)} = 1 \))
  - So our code can simply do this:
    \[
    J(X, y, \beta) = \frac{1}{2m} (X\beta - y)^2
    \]

In [10]: # Loss function for OLS regression (assumes X contains all 1s in its first column)
   
   def J(X, y, beta):
       return np.mean((X.dot(beta) - y) ** 2) / 2.0

Now let's find a model

In [11]: # Use pandas to read the CSV file
   
   df = pd.read_csv("datasets/dataset_corkA.csv")

   # Get the feature-values (just flarea) and the target values
   X = df["flarea"].values
   y = df["price"].values

   # Add the extra column to X
   X_augmented = add_dummy_feature(X)
# I invite you to modify these values
beta = np.array([800, -5])

# Calculate the loss
loss = J(X_augmented, y, beta)

# Then plot the training data and the model
fig = plt.figure()
plt.title("Training set and learned model")
xvals = np.array([[1, 0], [1, 500]])
plt.text(10, 900, "Loss: \(\text{"} + \text{str(loss)}\), color = "red")
plt.xlabel("Floors area (sq metres)"")
plt.ylabel("Price (000 euros)"")
plt.show()
In [14]:
# Use pandas to read the CSV file
df = pd.read_csv("datasets/dataset_corkA.csv")

# Get the feature-values (just bdrms and bthrms) and the target values
X = df["bdrms", "bthrms"].values
y = df["price"].values

# Add the extra column to X
X_augmented = add_dummy_feature(X)

In [15]:
# I invite you to modify these values
beta = np.array([100, 50, 50])

# Calculate the loss
loss = J(X_augmented, y, beta)

In [16]:
# Then plot the training data and the model
fig = plt.figure()
ax = Axes3D(fig)
ax.set_title("Training set and learned model")
ax.scatter(X[:,0], X[:,1], y, color = "green")
xvals = np.linspace(0, 10, 2)
yvals = np.linspace(0, 10, 2)
xxvals, yyvals = np.meshgrid(xvals, yvals)
ax.plot_surface(xxvals, yyvals, beta[0] + beta[1] * xxvals + beta[2] * yyvals, color=(0, 0, 1, 0.2))
ax.text(6, 14, 900, "Loss: " + str(loss), color = "red")
ax.set_xlabel("Bedrooms")
ax.set_xlim(0,10)
ax.set_ylabel("Bathrooms")
ax.set_ylim(0, 10)
ax.set_zlabel("Price (000 euros)")
ax.set_zlim(0, 1000)
plt.show()

- Keep modifying $\beta$ until you find the lowest loss
- We can't do a similar example with 3 or more features
  - Because we can't plot them
Finding OLS Models

- We've been trying out different values for $\beta$, looking for the model with lowest mean squared error by trial and error!
- In practice, it is not done by trial-and-error
- There are two main methods:
  - The normal equation (LinearRegression class in scikit-learn)
  - Various forms of gradient descent (SGDRegressor class in scikit-learn)
- We give a quick example of the first of these
  - (No need to add the extra column: the LinearRegression class does it for us)

```python
In [17]: # Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")

In [18]: # The features we want to select
numeric_features = ["flarea", "bdrms", "bthrms", "floors"]
nominal_features = ["type", "devment", "ber", "location"]

# Create the preprocessor
preprocessor = ColumnTransformer([ 
    ("num", StandardScaler(), numeric_features),
    ("nom", OneHotEncoder(handle_unknown="ignore"), nominal_features),
    remainder="drop" )

# Run the preprocessor
preprocessor.fit(df)
X = preprocessor.transform(df)

In [19]: # Create the estimator
linreg = LinearRegression()

In [20]: # Get the labels
y = df["price"].values

In [21]: # Fit the linear model
linreg.fit(X, y)

Out[21]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [22]: # Create your house
your_house_df = pd.DataFrame([ {
"flarea":126.0, 
"type":"semi-detached", 
"bdrms":3, 
"bthrms":1, 
"floors":2, 
"devment":"SecondHand", 
"ber":"B2", 
"location":"Glasheen"}])

# Transform it using the pipeline
your_house_transformed = preprocessor.transform(your_house_df)

# Predict its selling price
linreg.predict(your_house_transformed)

Out[22]: array([361.92216213])
```

- (In the next lecture, we will see how to include the linear regression step in the pipeline)