Solutions 1

Jumping Rivers

Object Oriented Programming

For the practical we are going to create a `Model` class which fits a linear regression model on initialisation and then add some methods for common things that we might wish to do with our regression models.

The code below will define a `Model` class which fits the model given an input, `X`, and response `y`.

```
class Model:
  def __init__(self,X,y):
    self.X = X
    self.y = y
    if len(X.shape) == 1:
      X = X.values.reshape(-1,1)
    X = np.c_[np.ones(X.shape[0]),X]
    xTx = np.dot(X.T,X)
    inverse_xTx = np.linalg.inv(xTx)
    xTy = np.dot(X.T,y)
    coef = np.dot(inverse_xTx,xTy)
    self.intercept = coef[0]
    self.gradients = coef[1:]
```

Our model class has 4 instance attributes

- `.X` the input predictor variables
- `.y` the output response variable
- `.intercept` the intercept of the fitted model
- `.gradients` the estimated gradients of the fitted model

1. First we will add a `predict` method to our model class. The following definition will perform a model prediction given an input `X`, intercept, `intercept` and gradients, `gradients`.

```
def predict(X, intercept,gradients):
  if len(X).shape == 1:
    X = X.values.reshape(-1,1)
  return intercept + np.dot(X,gradients)
```

Modify the function to use the instance attributes of the model and add it as a method to the `Model` class.

```
class Model:
  def __init__(self,X,y):
    self.X = X
    self.y = y
    if len(X.shape) == 1:
      X = X.values.reshape(-1,1)
    X = np.c_[np.ones(X.shape[0]),X]
    xTx = np.dot(X.T,X)
    inverse_xTx = np.linalg.inv(xTx)
    xTy = np.dot(X.T,y)
    coef = np.dot(inverse_xTx,xTy)
    self.intercept = coef[0]
    self.gradients = coef[1:]

  def predict(self,X):
    if len(X.shape) == 1:
      X = X.values.reshape(-1,1)
    return self.intercept + np.dot(X,self.gradients)
```

2. Add magic methods to the class that allows `print(model)` and `model` to show the intercept and gradients of the fitted model

```
def __str__(self):
  return 'A linear regression model with parameters \n \
  \n Intercept: {} \
  \n Gradients: {}'.format(self.intercept, self.gradients[:,0])

def __repr__(self):
  return self.__str__()
```

3. Update your initialisation method to automatically calculated the fitted values and residuals of the model and store them as instance attributes. Fitted values are calculated by predicting from the model using the original `X` inputs. Residuals are defined as the fitted values - observed response.

```
def __init__(self,X,y):
  self.X = X
  self.y = y
  if len(X.shape) == 1:
    X = X.values.reshape(-1,1)
  X = np.c_[np.ones(X.shape[0]),X]
  xTx = np.dot(X.T,X)
  inverse_xTx = np.linalg.inv(xTx)
  xTy = np.dot(X.T,y)
  coef = np.dot(inverse_xTx,xTy)
  self.Xmat = X
  self.intercept = coef[0]
  self.gradients = coef[1:]
  # add in exercise
  self.fitted = self.predict(self.X)
  self.residuals = self.fitted - self.y
```

4. Create a new class which inherits from `Model`. Add a `plot_one` method which could be used to show a scatter plot of the response against a chosen input, together with a line to show the fitted model.

```
class plottableModel(Model):
  def plot_one(self,dim = 0):
    cols = self.X.shape[1]
    if dim > (cols - 1):
      raise Exception('You chose column {} (index {}) but there are \
      only {} columns to choose from.'.format(
        dim-1,dim,cols
      ))
    df = self.X.assign(y = self.y, fitted = self.fitted)
    df.sort_values(df.columns[dim],inplace = True)
    p = df.plot.scatter(x=dim,y = 'y')
    df.plot.line(x = dim, y = 'fitted', ax = p, c = 'black')
    plt.show()
```

5. Add a class method to your newsub class that allows you to take an existing instance of the original `Model` class and create and instance of the new subclass.

```
@classmethod
def make_plottable(cls,m):
  return cls(m.X,m.y)

# For reference, both full classes
import numpy as np
import matplotlib.pyplot as plt
class Model:
  def __init__(self,X,y):
    self.X = X
    self.y = y
    if len(X.shape) == 1:
      X = X.values.reshape(-1,1)
    X = np.c_[np.ones(X.shape[0]),X]
    xTx = np.dot(X.T,X)
    inverse_xTx = np.linalg.inv(xTx)
    xTy = np.dot(X.T,y)
    coef = np.dot(inverse_xTx,xTy)
    self.Xmat = X
    self.intercept = coef[0]
    self.gradients = coef[1:]
    self.fitted = self.predict(self.X)
    self.residuals = self.fitted - self.y

  def predict(self,X):
    if len(X.shape) == 1:
      X = X.values.reshape(-1,1)
    return self.intercept + np.dot(X,self.gradients)

  def __str__(self):
    return 'A linear regression model with parameters \n \
    \n Intercept: {} \
    \n Gradients: {}'.format(self.intercept, self.gradients[:,0])

  def __repr__(self):
    return self.__str__()
class plottableModel(Model):
  def plot_one(self,dim = 0):
    cols = self.X.shape[1]
    if dim > (cols - 1):
      raise Exception('You chose column {} (index {}) but there are \
      only {} columns to choose from.'.format(
        dim-1,dim,cols
      ))
    df = self.X.assign(y = self.y, fitted = self.fitted)
    df.sort_values(df.columns[dim],inplace = True)
    p = df.plot.scatter(x=dim,y = 'y')
    df.plot.line(x = dim, y = 'fitted', ax = p, c = 'black')
    plt.show()

  @classmethod
  def make_plottable(cls,m):
    return cls(m.X,m.y)
```