CS4618: Artificial Intelligence I

Linear Models

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

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.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import add_dummy_feature

from sklearn.linear_model import LinearRegression

from mpl_toolkits.mplot3d import Axes3D

# Class, for use in pipelines, to select certain columns from a DataFram
e and convert to a numpy array
# From A. Geron: Hands-On Machine Learning with Scikit-Learn & TensorFlo
w, O'Reilly, 2017
# Modified by Derek Bridge to allow for casting in the same ways as pand
as.DatFrame.astype
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names, dtype=None):
        self.attribute_names = attribute_names
        self.dtype = dtype
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X_selected = X[self.attribute_names]
        if self.dtype:
            return X_selected.astype(self.dtype).values
        return X_selected.values

# Class, for use in pipelines, to binarize nominal-valued features (whil
e avoiding the dummy variabe trap)
# By Derek Bridge, 2017
class FeatureBinarizer(BaseEstimator, TransformerMixin):
    def __init__(self, features_values):
        self.features_values = features_values
        self.num_features = len(features_values)
        self.labelencodings = [LabelEncoder().fit(feature_values) for fe
ature_values in features_values]
        self.onehotencoder = OneHotEncoder(sparse=False,
            n_values=[len(feature_values) for feature_values in features
_values])
        self.last_indexes = np.cumsum([len(feature_values) - 1 for featu
re_values in self.features_values])
    def fit(self, X, y=None):
        for i in range(0, self.num_features):
            X[:, i] = self.labelencodings[i].transform(X[:, i])
        return self.onehotencoder.fit(X)
    def transform(self, X, y=None):
        for i in range(0, self.num_features):
            X[:, i] = self.labelencodings[i].transform(X[:, i])
        onehotencoded = self.onehotencoder.transform(X)
        return np.delete(onehotencoded, self.last_indexes, axis=1)
    def fit_transform(self, X, y=None):
        onehotencoded = self.fit(X).transform(X)
        return np.delete(onehotencoded, self.last_indexes, axis=1)
    def get_params(self, deep=True):
        return {"features_values" : self.features_values}
    def set_params(self, **parameters):
        for parameter, value in parameters.items():
            self.setattr(parameter, value)
        return self
```

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. $y = 3 + 14x_1x_2 + 12x_3$
- Evalute $y = 2 + 3x_1 + 4x_2 + 5x_3$:

  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 β_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 $\boldsymbol{x} = [x_1, x_2, x_3]$ and $\boldsymbol{\beta} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$
  - 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:

$$y = \beta_0 + \sum_{i=1}^{n} \boldsymbol{\beta}_i \boldsymbol{x}_i$$

- It can also, equivalently, be written in this form:

$$y = \beta_0 + \boldsymbol{x}\boldsymbol{\beta}$$

- Hence, to evaluate a linear equation, simply multiply the two vectors and add β_0

Evaluating a linear equation in numpy

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

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

y
```

Out[4]: 145

- But you don't need to write your own loop: use numpy library's matrix multiplication method

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 β_0) into a column vector 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 $\boldsymbol{x} = [1, x_1, x_2, x_3]$ and $\boldsymbol{\beta} = \begin{bmatrix} 12 \\ 3 \\ 4 \\ 5 \end{bmatrix}$
  - 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:

$$y = \sum_{i=0}^{n} \boldsymbol{\beta}_i \boldsymbol{x}_i$$

- It can also, equivalently, be written in this form:

$$y = \boldsymbol{x}\boldsymbol{\beta}$$

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

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 $\boldsymbol{x}$
  - 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, $\boldsymbol{X}$, but with an extra element $\boldsymbol{x}_0^{(i)}$ in each row i, all of which will be 1, then we can obtain all the results by simple matrix multiplication:

$$y = \boldsymbol{X\beta}$$

  - E.g.

$$\boldsymbol{y} = \begin{bmatrix} 1 & 7 & 3 & 20 \\ 1 & 10 & 20 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 100 & 0 & -2 \end{bmatrix} \begin{bmatrix} 12 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

  It produces a vector of results, e.g. $\boldsymbol{y} = \begin{bmatrix} 145 \\ 122 \\ 24 \\ 302 \end{bmatrix}$

Evaluating a linear equation multiple times in numpy

- Same story: no loop, use matrix mutliplication

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]))
y
```

Out[7]:
```
array([145, 122,  24, 302])
```

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

Linear Models

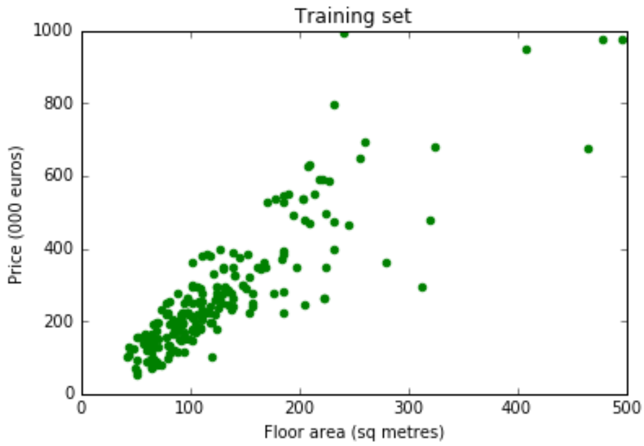
- Recall: We want to learn a model from a labeled training set
- For the remainder of *CS4618* (but not *CS4619*), we will content ourselves with learning a linear model
  - In regression, we'll try to find a linear equation that best fits the training examples
  - In classification, we'll try to find 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 (in stats-speak: 'univariate')

Linear Regression: Univariate

- 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 pipeline
- 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:
$$\hat{y} = \beta_0 + \beta_1 \times flarea$$
- In other words, our goal is to choose values for β_0 and β_1
  - From the point of view of plotting this line, what's β_0? What's β_1?
  - E.g. we could choose $\beta_0 = 800$ and $\beta_1 = -5$
  - Or we could choose $\beta_0 = 200$ and $\beta_1 = 5$

  Lets' refer to any particular choice as $h_{\boldsymbol{\beta}}$ (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 β_0 and β_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 $h_{\boldsymbol{\beta}}$ and gives it a score
    - Low numbers are better!
  - For each $\boldsymbol{x}$ in the training set, it will compare $h_{\boldsymbol{\beta}}(\boldsymbol{x})$, which is the *prediction* that $h_{\boldsymbol{\beta}}$ makes on $\boldsymbol{x}$, with the *actual* value y
- The loss function most usually used for linear regression is the **mean squared error**:
  - The difference between the prediction and the actual value, squared
  - But averaged over all the examples in the training set

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

  - 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:

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

— the 'winner' is still the same, but this makes the calculus 'tidier' later

The loss function in numpy

- Looks like a loop: work out $h_{\boldsymbol{\beta}}$ for each $\boldsymbol{x}^{(i)}$
  - But $h_{\boldsymbol{\beta}}$ is a linear equation, and we want to evaluate it lots of times (for each example $\boldsymbol{x}^{(i)}$)
  - So we use the vectorized approach from above (assuming all the examples contain an extra element, $\boldsymbol{x}_0^{(i)} = 1$)
- So our code can simply do this:

$$J(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\beta}) = \frac{1}{2m}(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})^2$$

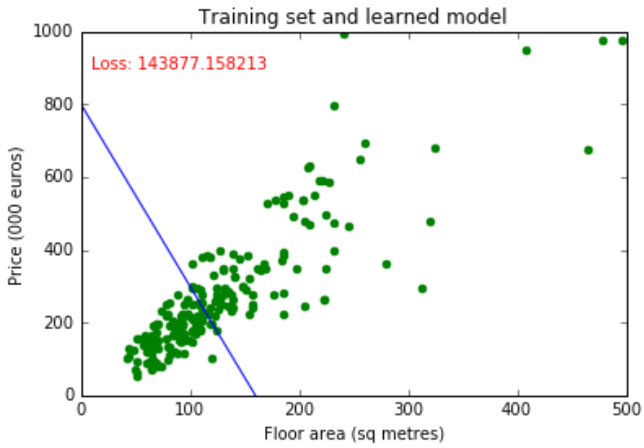
In [10]:
```
# Loss function for OLS regression (assumes X contains all 1s in its fir
st column)
def J(X, y, beta):
    return np.mean((X.dot(beta) - y) ** 2) / 2.0
```

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)
```

In [12]:
```
# I invite yout to modify these values
beta = np.array([800, -5])

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

In [13]:
```
# Then plot the training data and the model
fig = plt.figure()
plt.title("Training set and learned model")
plt.scatter(X, y, color = "green")
xvals = np.array([[1, 0], [1, 500]])
plt.plot(xvals, xvals.dot(beta), color = "blue")
plt.text(10, 900, "Loss: " + str(loss), color = "red")
plt.xlabel("Floor area (sq metres)")
plt.xlim(0, 500)
plt.ylabel("Price (000 euros)")
plt.ylim(0, 1000)
plt.show()
```



- Keep modifying β until you find the lowest loss

Linear Regression: Multivariate

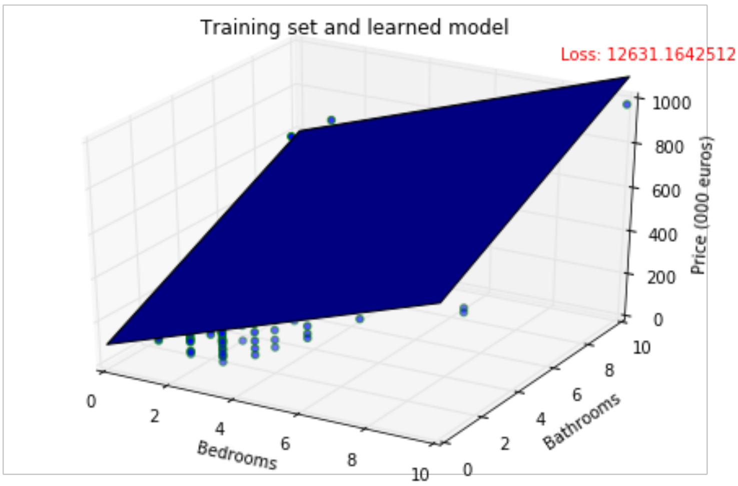
- We considered only one feature ($flarea$)
  - This enabled easy visualisation on a 2D plot
  - The model is a straight line
- The only differences when we move to more than one feature (stats-speak: multivariate):
  - We can't plot so easily
  - The model is a plane (when there are two features)
  - The model is a hyperplane (when there are more than two features)
- *All the maths and the Python for the loss function remain the same*

Now let's find a model using two features

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] * y
yvals)
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 $\boldsymbol{\beta}$ 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 β, 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)

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 pipelines
numeric_pipeline = Pipeline([
        ("selector", DataFrameSelector(numeric_features))
    ])

nominal_pipeline = Pipeline([
        ("selector", DataFrameSelector(nominal_features)),
        ("binarizer", FeatureBinarizer([df[feature].unique() for feature
in nominal_features]))])

pipeline = Pipeline([("union", FeatureUnion([("numeric_pipeline", numeri
c_pipeline),
                                             ("nominal_pipeline", nomina
l_pipeline)]))])
```

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

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

In [21]:
```
# Run the pipeline to prepare the data
pipeline.fit(df)
X = pipeline.transform(df)
```

In [22]:
```
# Fit the linear model
linreg.fit(X, y)
```

Out[22]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=Fal
se)
```

In [23]:
```
# Create your house
your_house_df = pd.DataFrame([{"flarea":114.0, "type":"Semi-detached", "
bdrms":3, "bthrms":2, "floors":2,
                                "devment":"SecondHand", "ber":"B2", "loca
tion":"Glasheen"}])

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

# Predict its selling price
linreg.predict(your_house_scaled)
```

Out[23]:
```
array([ 270.2835876])
```

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

In [ ]: