Supervised Machine Learning

Notes by José A. Espiño P. [1]

Summer Semester 2022–2023

[1] The content in these notes is sourced from what was covered in the MOOG the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Introduction to Machine Learning

Machine Learning is a field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel). The more options you give to a learning algorithm, the better they will perform. The two main types of ML algorithms are supervised learning and unsupervised learning; the former is the one that has experienced the most rapid advancements and been used the most.

Supervised Learning refers to algorithms that learn input to output mappings. The key characteristic of these algorithms is that they are given examples to learn from, namely, the correct label y for a given input x. By seeing a large amount of these pairs, the algorithm eventually learns how to match a given x to a satisfactory y by generating an equation (e.g. for a straight line, a curve) with appropriate values that will allow the prediction to be accurate. Within supervised learning, there are two big types: regression, which maps the input to continuous values in a range, and classification, which maps x to a discrete set of possible outputs.

Unsupervised Learning refers to finding patterns or structures in data as opposed to classifying (*supervising*) it. Clustering is a subtype of unsupervised learning that, as the name implies, places unlabeled data in different clusters. Anomaly Detection detects unsual events or features within a group of data. Dimensionality Reduction compresses the size of a set of data.

Some notation:

- Training Set: data used to train the model. The total number of training samples is denoted by m
- Input Feature: input value(s). Denoted as x
- Target Variable: output value(s). Denoted as y
- Parameters of a model: variables you adjust during training to improve the model. Also called coefficients or weights.

2 Linear Regression Model

It consists of fitting a straight line to the data and is one of the most widely used ML algorithms. Recall that in a training set we have input-output pairs. To train the model, you feed this set to the algorithm so that it produces a function f, which then can produce an output given a new input(which is not found in the training set). This estimated output is normally denoted as $\hat{y}$.
In linear regression, f will be a linear function, namely: $f_{w,b}(X) = wx + b$. The values chosen for w and b are the ones that will determine $\hat{y}$. Linear functions tend to be used due to their simplicity: this makes manipulation easier or can be used as a base for fitting more complex non–linear models. Linear Regression can be univariate or multiple depending on the amount of input features introduced.
In order for the algorithm to work, we have to construct a *cost function*. This function allows us to measure how well the line fits the data; it takes the prediction $\hat{y}$ and compares it to the target y by computing $(\hat{y}^i - y^i)^2$ (the error). We want to measure the error accross the entire dataset, so the function will be:

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^i - y^i)^2$$

Or

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^i) - y^i)^2$$

This is called the *squared error cost function*, which is one of many possible cost functions to use. Our goal is to find the parameters that successfully minimise the cost function we have chosen for our model.
This function can be implemented in Python as follows:

```
import numpy as np
#the following arrays contain the training data
x_train = np.array([1.0, 2.0])           #(size in 1000 square feet)
y_train = np.array([300.0, 500.0])           #(price in 1000s of
    dollars)

def compute_cost(x, y, w, b):
    """
    Computes the cost function for linear regression.

    Args:
      x (ndarray (m,)): Data, m examples
      y (ndarray (m,)): target values
      w,b (scalar)    : model parameters

    Returns
        total_cost (float): The cost of using w,b as the parameters
    for linear regression
              to fit the data points in x and y
    """
    # number of training examples
    m = x.shape[0]
```

```
    cost_sum = 0
    for i in range(m):
        f_wb = w * x[i] + b
        cost = (f_wb - y[i]) ** 2
        cost_sum = cost_sum + cost
    total_cost = (1 / (2 * m)) * cost_sum

    return total_cost

cost = compute_cost(x_train, y_train, parameterw, parameterb)
return cost
```

How can we minimise the cost function (and any other function for that matter) then? We can utilise *gradient descent*:

1. Start off with random guesses for the initial values of the variables you are trying to minimise
2. Update the values of the variables repeatedly until the cost function settles at a minimum (reaches convergence). This update is done through the following:

$$w = w - \alpha \frac{\partial}{\partial w} J(w,b) = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^i - y^i)) x^i$$

and

$$b = b - \alpha \frac{\partial}{\partial b} J(w,b) = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^i - y^i))$$

Where α is the learning rate.
It is important to note that the variables must be updated **simultaneously** on each iteration.

The choice of the learning rate (α) is very important! If it is too small, an expensive number of iterations will be needed to reach convergence. Alternatively, if it is too big, it might overshoot and never reach convergence. The learning rate can be scheduled: be big in the first iterations and become smaller as the algorithm gets closer to the minimum. If the algorithm takes too long or the cost value increases at any point, a good way to troubleshoot is by decreasing the value of the learning rate. A technique commonly used is to set a possible range for the learning rate and start from the smallest number to the largest in the range (until the cost function starts to increase). This will eventually provide us with the ideal learning rate.
An interesting property of gradient descent is that it will always lead you to the local minimum closest to the initial values it is given.
Take a look at a sample implementation:

```
import math, copy
import numpy as np

# Load our data set
x_train = np.array([1.0, 2.0])   #features
```

```
y_train = np.array([300.0, 500.0])   #target value

#Function to calculate the cost
def compute_cost(x, y, w, b):

    m = x.shape[0]
    cost = 0

    for i in range(m):
        f_wb = w * x[i] + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost

    return total_cost

def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray (m,)): Data, m examples
      y (ndarray (m,)): target values
      w,b (scalar)    : model parameters
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameter b
     """

    # Number of training examples
    m = x.shape[0]
    dj_dw = 0
    dj_db = 0

    for i in range(m):
        f_wb = w * x[i] + b
        dj_dw_i = (f_wb - y[i]) * x[i]
        dj_db_i = f_wb - y[i]
        dj_db += dj_db_i
        dj_dw += dj_dw_i
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_dw, dj_db

def gradient_descent(x, y, w_in, b_in, alpha, num_iters, cost_function, gradient_function):
    """
    Performs gradient descent to fit w,b. Updates w,b by taking
    num_iters gradient steps with learning rate alpha

    Args:
      x (ndarray (m,))  : Data, m examples
      y (ndarray (m,))  : target values
```

```
      w_in,b_in (scalar): initial values of model parameters
      alpha (float):      Learning rate
      num_iters (int):    number of iterations to run gradient descent
      cost_function:      function to call to produce cost
      gradient_function: function to call to produce gradient

    Returns:
      w (scalar): Updated value of parameter after running gradient descent
      b (scalar): Updated value of parameter after running gradient descent
      """

    b = b_in
    w = w_in

    for i in range(num_iters):
        # Calculate the gradient and update the parameters using gradient_function
        dj_dw, dj_db = gradient_function(x, y, w , b)

        # Update Parameters
        b = b - alpha * dj_db
        w = w - alpha * dj_dw

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters/10) == 0:
            print(f"Iteration {i:4}: Cost {J_history[-1]:0.2e} ",
                  f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e}  ",
                  f"w: {w: 0.3e}, b:{b: 0.5e}")

    return w, b #return w and J

# initialize parameters
w_init = 0
b_init = 0

# some gradient descent settings
iterations = 10000
tmp_alpha = 1.0e-2

# run gradient descent
w_final, b_final= gradient_descent(x_train ,y_train, w_init, b_init, tmp_alpha,
                                                              iterations, compute_cost, compute_gradient)
print(f"(w,b) found by gradient descent: ({w_final:8.4f},{b_final:8.4f})")
```

3 Multiple Linear Regression

In most cases there is more than one element as an input. In that instance, every element in the input training set is usually represented as a row vector with n elements, where n =the number of input features. The linear model will be altered to fit these new features, from $f_{w,b}(X) = w\,x + b$ to $f_{w,b}(X) = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$. A simpler definition can be obtained by letting W be a vector containing w_1 through w_n. We do the same with X, thus obtaining: $f_{\vec{w},b}(\vec{X}) = \vec{w} \cdot \vec{x} + b$
Vectorisation is a concept often used in ML; it can make the code both more efficient and short. A common way to implement vectorisation is by using functions defined in the `NumPy` library. For instance, as opposed to writing a for–loop to iterate through two vectors and sum the product of each pair, we can just use the function `np.dot(v1,v2)`. Some other common vectorisation methods are:

- `outer(v1,v2)`: computes the outer product of two vectors
- `multiply(v1,v2)`: matrix product of two arrays
- `zeros((n,m))`: returns a matrix of given shape and type filled with zeroes
- `vector.shape`: returns the shape of the element
- `vector[index]`: returns item at index
- `matrix[x,y]`: accesses item at coordinates x,y

Vectorisation is more efficient because it uses the computer's parallel processing hardware. Similarly to the linear model, gradient descent will also undergo modifications to allow for multiple input features. For n features ($n \geq 2$):
$repeat\ \{$

$$w_1 = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{\,i} - y^{\,i}))x_1^i$$

...

$$w_n = w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{\,i} - y^{\,i}))x_n^i$$

And then, as before, we update b:

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{\,i} - y^{\,i}))$$

Multiple Variable Linear Regression can be implemented with Python as follows:

```
import copy, math
import numpy as np

#sample input
X_train = np.array([[2104, 5, 1, 45], [1416, 3, 2, 40], [852, 2, 1,
    35]])
y_train = np.array([460, 232, 178])

```

```
#Initialisation with random variables
b_init = 785.1811367994083
w_init = np.array([ 0.39133535, 18.75376741, -53.36032453,
    -26.42131618])

def predict(x, w, b):
    """
    single predict using linear regression
    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):             model parameter

    Returns:
      p (scalar):  prediction
    """
    p = np.dot(x, w) + b
    return p

def compute_cost(X, y, w, b):
    """
    compute cost
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      cost (scalar): cost
    """
    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b           #(n,)(n,) = scalar (
    see np.dot)
        cost = cost + (f_wb_i - y[i])**2       #scalar
    cost = cost / (2 * m)                      #scalar
    return cost
def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the
    parameters w.
      dj_db (scalar):       The gradient of the cost w.r.t. the
    parameter b.
    """
```

```
    m,n = X.shape           #(number of examples, number of features)
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw

def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn w and b. Updates w and b by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X (ndarray (m,n))   : Data, m examples with n features
      y (ndarray (m,))    : target values
      w_in (ndarray (n,)) : initial model parameters
      b_in (scalar)       : initial model parameter
      cost_function       : function to compute cost
      gradient_function   : function to compute the gradient
      alpha (float)       : Learning rate
      num_iters (int)     : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,)) : Updated values of parameters
      b (scalar)       : Updated value of parameter
      """



    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        dj_db,dj_dw = gradient_function(X, y, w, b)   ##None

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw               ##None
        b = b - alpha * dj_db               ##None


```

```
107
108         # Print cost every at intervals 10 times or as many
    iterations if < 10
109         if i% math.ceil(num_iters / 10) == 0:
110             print(f"Iteration {i:4d}: Cost {J_history[-1]:8.2f}   ")
111
112     return w, b #return final w,b
113
114 # initialize parameters
115 initial_w = np.zeros_like(w_init)
116 initial_b = 0.
117 # some gradient descent settings
118 iterations = 1000
119 alpha = 5.0e-7
120 # run gradient descent
121 w_final, b_final = gradient_descent(X_train, y_train, initial_w,
    initial_b,
122                                                     compute_cost,
    compute_gradient,
123                                                     alpha,
    iterations)
124 print(f"b,w found by gradient descent: {b_final:0.2f},{w_final} ")
125 m,_ = X_train.shape
126 for i in range(m):
127     print(f"prediction: {np.dot(X_train[i], w_final) + b_final:0.2f
    }, targ
```

Feature Scaling is a technique that lets gradient descent run much faster. When the possible range of a feature is large, the model is likely to learn to choose a relatively small parameter value; conversely, when the possible range of a feature is small, its parameter will likely take a large value. When the range of certain features is too disimillar, it takes a long while for gradient descent to reach the minimum of the cost function. To work around this, we *scale* some features so that their range is the same (usually from 0 to 1). This leads to a quicker path to the minimum.
Feature scaling can be implemented by dividing each entry by the maximum value in the range of the feature. In addition to this, feature scaling can also be implemented through normalisation: re–scaling all features so that they are centered around zero (having both positive and negative values). To calculate it, you must first find the average (μ). Then, for each entry, you will do $x_i = \frac{x_i - \mu}{\text{maximum value - minimum value}}$. The last commonly–used method is Z–score normalisation: you calculate the mean (μ) as well as the standard deviation (σ). Then, each value x_i will be $x_i = \frac{x_i - \mu}{\sigma}$
As a rule of thumb, when doing feature scaling, you should aim for ranges somewhere around -1 and $+1$.
Take a look at this implementation of Z–score normalisation in Python:

```
1 def zscore_normalize_features(X):
2     """
3     computes  X, zcore normalized by column
4
5     Args:
6       X (ndarray (m,n))     : input data, m examples, n features
```

```
    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (n,))  : standard deviation of each feature
    """
    # find the mean of each column/feature
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)
```

If the value of the cost function J ever increases through the execution of gradient descent, either you have made a poor choice of a learning rate, or there is a bug in the code. Usually, convergence is detected by setting a very small value (usually we have $\epsilon = 10^{-3}$) and checking if the cost function decreases by an amount smaller than that value in one iteration. If that is the case, we can stop the gradient descent.

3.1 Polynomial Regression

Sometimes a straight line is not the best fit for the dataset. When that is the case, we may duplicate certain parameters and exponentiate those parameters. When this is the case, feature scalling is as important as ever — this makes the range of the parameter grow exponentially.

3.2 Scikit–Learn

Scikit–Learn is a very popular ML Python library. Take a look at the code excerpt below to get familiar with some common methods that are used in implementing gradient descent:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler #performs z-score normalisation
from lab_utils_multi import  load_house_data #only used in this lab
from lab_utils_common import dlc
np.set_printoptions(precision=2)
plt.style.use('./deeplearning.mplstyle')

X_train, y_train = load_house_data()
X_features = ['size(sqft)','bedrooms','floors','age']

#scaling
```

```
scaler = StandardScaler()
X_norm = scaler.fit_transform(X_train)

#create regression model
sgdr = SGDRegressor(max_iter=1000)
sgdr.fit(X_norm, y_train)

#view parameters
b_norm = sgdr.intercept_
w_norm = sgdr.coef_

# make a prediction using sgdr.predict()
y_pred_sgd = sgdr.predict(X_norm)
# make a prediction using w,b.
y_pred = np.dot(X_norm, w_norm) + b_norm
```

4 Classification

Binary classification is when there are only two possible output categories (e.g. spam email detectors). Linear regression does not work for classification problems because any outlier training samples may cause the *decision boundary* to shift and thus render the algorithm inaccurate. Classification is instead approached with algorithms such as *logistic regression*. **Logistic Regression** is one of the most commonly used classification algorithms currently. With logistic regression, our goal is to fit a curve onto the data to make the output be one of a discrete set of possible values. The meat of logistic regression is the **Sigmoid function**, also known as the logistic function. The Sigmoid function ($g(z) = \frac{1}{1+e^{-z}}$) will only output values between 0 and 1. That considered, logistic regression can be achieved through the following steps:

1. Let us define the basis of the model we shall be using; $z = \vec{w} \cdot \vec{x} + b$
2. Pass z into the Sigmoid function
3. This gives us the Logistic Regression model:

$$f_{\vec{w},b}(\vec{X}) = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$

The output of the logistic regression model is the probability that the inputted value is part or not of a given category. A way we can let the algorithm predict an output is to set a threshold; every value above it will be of a certain category, every value below it will be in a different category. The *decision boundary* is the line where $z = \vec{w} \cdot \vec{x} + b = 0$, which separates elements of different categories. Using polynomials, you can make the decision boundary be a complex non–linear function.
In logistic regression, the **cost function** is different to the one we utilised earlier; this is because using the squared error cost function on the logistic regression model, we obtain a non-convex graph. This implies that gradient descent can get sucked into one of the many

local minima to be found in the function. The squared error cost function we have been using so far is $J(\vec{w},b)=\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(f_{\vec{w},b}(\vec{x}^i)-y^i)^2$. If we regard $L=f_{\vec{w},b}(\vec{x}^i)-y^i)^2$ as the *loss function* of the equation, we can try change it to make the squared error cost be convex too in the case of logistic regression. The loss function that achieves this is:

$$L(f_{\vec{w},b(\vec{X}^i),y^i})=\begin{cases}-log(f_{\vec{w},b}(\vec{X}^i)), & y^i=1,\\ -log(1-f_{\vec{w},b}(\vec{X}^i)), & y^i=0.\end{cases}$$

This loss function can be further simplified to allow for higher efficiency. This results in:

$$L(f_{\vec{w},b}(\vec{X}^i),y^i)=-y^i log(f_{\vec{w},b}(\vec{X}^i)-(1-y^i)log(1-f_{\vec{w}},b)(\vec{X}^{(i)}))$$

This considered, take a look at how this new cost function can be implemented on Python:

```python
import numpy as np

def compute_cost_logistic(X, y, w, b):
    """
    Computes cost

    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      cost (scalar): cost
    """

    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        z_i = np.dot(X[i],w) + b
        f_wb_i = sigmoid(z_i)
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)

    cost = cost / m
    return cost
```

With all of this covered, we can proceed to implement gradient descent: similarly to linear regression, gradient descent will consist in finding the parameters that can minimise the cost function J by continuously performing the following until convergence:

$$w_1 = w_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))x_1^i$$

...

$$w_n = w_n - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))x_n^i$$

And then, as before, we update b:

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))$$

As you might have noticed, this is the same as the gradient descent formula in the case of linear regression. The key difference here is that our definition of $f_{\vec{w},b}$ has changed: it is the Sigmoid function in logistic regression. This can all be implemented with Python as follows:

```
import copy, math
import numpy as np

#random dataset for the example
X_train = np.array([[0.5, 1.5], [1,1], [1.5, 0.5], [3, 0.5], [2, 2],
    [1, 2.5]])
y_train = np.array([0, 0, 0, 1, 1, 1])

def compute_gradient_logistic(X, y, w, b):
    """
    Computes the gradient for linear regression

    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
    Returns
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the
    parameters w.
      dj_db (scalar)      : The gradient of the cost w.r.t. the
    parameter b.
    """
    m,n = X.shape
    dj_dw = np.zeros((n,))                           #(n,)
    dj_db = 0.

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)          #(n,)(n,)=
    scalar
        err_i  = f_wb_i  - y[i]                       #scalar
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]      #scalar
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m                                   #(n,)
    dj_db = dj_db/m                                   #scalar

    return dj_db, dj_dw

def gradient_descent(X, y, w_in, b_in, alpha, num_iters):
    """
    Performs batch gradient descent

    Args:
      X (ndarray (m,n)   : Data, m examples with n features
      y (ndarray (m,))   : target values
      w_in (ndarray (n,)): Initial values of model parameters
      b_in (scalar)      : Initial values of model parameter
      alpha (float)      : Learning rate
      num_iters (scalar) : number of iterations to run gradient
```

```
    descent

    Returns:
      w (ndarray (n,))    : Updated values of parameters
      b (scalar)          : Updated value of parameter
    """

    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):
        # Calculate the gradient and update the parameters
        dj_db, dj_dw = compute_gradient_logistic(X, y, w, b)

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters / 10) == 0:
            print(f"Iteration {i:4d}: Cost {J_history[-1]}   ")

    return w, b        #return final w,b
```

5 Overfitting and Underfitting

Underfitting refers to when the model is too general and does not provide the specificity required by the dataset provided. Underfitting is also referred to as the algorithm having high bias. Conversely, if the model is too specific to the training set, fitting the data too well, it is referred to as overfitting, or that the model has high variance. These two problems are relevant because they prevent the model from achieving generalisation: their predictions will not be accurate when fed new inputs.

There are several ways to address a model with high variance. Adding more training data is a very efficient method; however, data is not always readily available. Another method is to select features to include or exclude; this works because sometimes some features might not provide enough data, which skews the model. The third technique used to tackle overfitting is regularisation, namely, reducing the size of parameters w_j without demanding them be zero. Regularisation lets you keep all of your features but prevents them from having an overly large effect.

A quick way to achieve regularisation is by altering the cost function. For instance, we can multiply the weight we want to penalise by a very large number and sum it to the cost function. This means that the weight will be "penalised" and will be prevented from growing too large when conducting gradient descent. Very often, we do not know which parameters will be problematic. When that is the case, we penalise all of them by adding a term $\frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$. The value λ is known as the regularisation parameter, which we have to choose every time

we apply regularisation. Usually the term b is not penalised!
Since regularisation implies altering the cost function, this will also have an impact on the gradient descent algorithm:
The case of linear regression now looks like
For n features ($n \geq 2$):
$repeat$ {

$$w_1 = w_1 - \alpha[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))x_1^i + \frac{\lambda}{m}w_1]$$

...

$$w_n = w_n - \alpha[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))x_n^i + \frac{\lambda}{m}w_n]$$

And then, as before, we update b:

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))$$

In the case of logistic regression, it will look like:

$$w_1 = w_1 - \alpha[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))x_1^i + \frac{\lambda}{m}w_n]$$

...

$$w_n = w_n - \alpha[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))x_n^i + \frac{\lambda}{m}w_n]$$

And then, as before, we update b:

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^i - y^i))$$

Let us see an implementation of logistic regression with regularisation on Python:

```
import numpy as np
import copy
import math

mapped_X =  map_feature(X_train[:, 0], X_train[:, 1])
# load dataset
X_train, y_train = load_data("data/ex2data2.txt")

def compute_cost_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the cost over all examples
    Args:
      X : (array_like Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : (array_like Shape (n,)) Values of bias parameter of the
    model
      lambda_ : (scalar, float)    Controls amount of regularization
    Returns:
      total_cost: (scalar)         cost
    """
```

```
21
22      m, n = X.shape
23
24      # Calls the compute_cost function that you implemented above
25      cost_without_reg = compute_cost(X, y, w, b)
26
27      # You need to calculate this value
28      reg_cost = sum(np.square(w))
29
30      ### START CODE HERE ###
31
32      ### END CODE HERE ###
33
34      # Add the regularization cost to get the total cost
35      total_cost = cost_without_reg + (lambda_/(2 * m)) * reg_cost
36
37      return total_cost
38
39  X_mapped = map_feature(X_train[:, 0], X_train[:, 1])
40  np.random.seed(1)
41  initial_w = np.random.rand(X_mapped.shape[1]) - 0.5
42  initial_b = 0.5
43  lambda_ = 0.5
44  cost = compute_cost_reg(X_mapped, y_train, initial_w, initial_b,
        lambda_)
45
46  print("Regularized cost :", cost)
47
48  # UNIT TEST
49  compute_cost_reg_test(compute_cost_reg)
50
51  def compute_gradient_reg(X, y, w, b, lambda_ = 1):
52      """
53      Computes the gradient for linear regression
54
55      Args:
56        X : (ndarray Shape (m,n))   variable such as house size
57        y : (ndarray Shape (m,))    actual value
58        w : (ndarray Shape (n,))    values of parameters of the model
59        b : (scalar)                value of parameter of the model
60        lambda_ : (scalar,float)    regularization constant
61      Returns
62        dj_db: (scalar)             The gradient of the cost w.r.t.
        the parameter b.
63        dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t.
        the parameters w.
64
65      """
66      m, n = X.shape
67
68      dj_db, dj_dw = compute_gradient(X, y, w, b)
69
70      ### START CODE HERE ###
71      for i in range(n):
```

```
        dj_dw[i] = dj_dw[i] + (lambda_/m )* w[i]
    ### END CODE HERE ###

    return dj_db, dj_dw

X_mapped = map_feature(X_train[:, 0], X_train[:, 1])
np.random.seed(1)
initial_w  = np.random.rand(X_mapped.shape[1]) - 0.5
initial_b = 0.5

lambda_ = 0.5
dj_db, dj_dw = compute_gradient_reg(X_mapped, y_train, initial_w, initial_b, lambda_)

print(f"dj_db: {dj_db}", )
print(f"First few elements of regularized dj_dw:\n {dj_dw[:4].tolist()}", )

# UNIT TESTS
compute_gradient_reg_test(compute_gradient_reg)

# Initialize fitting parameters
np.random.seed(1)
initial_w = np.random.rand(X_mapped.shape[1])-0.5
initial_b = 1.

# Set regularization parameter lambda_ (you can try varying this)
lambda_ = 0.01

# Some gradient descent settings
iterations = 10000
alpha = 0.01

w,b, J_history,_ = gradient_descent(X_mapped, y_train, initial_w, initial_b,
                                    compute_cost_reg, compute_gradient_reg,
                                    alpha, iterations, lambda_)

p = predict(X_mapped, w, b)

print('Train Accuracy: %f'%(np.mean(p == y_train) * 100))
```