Machine Learning, Spring 2020

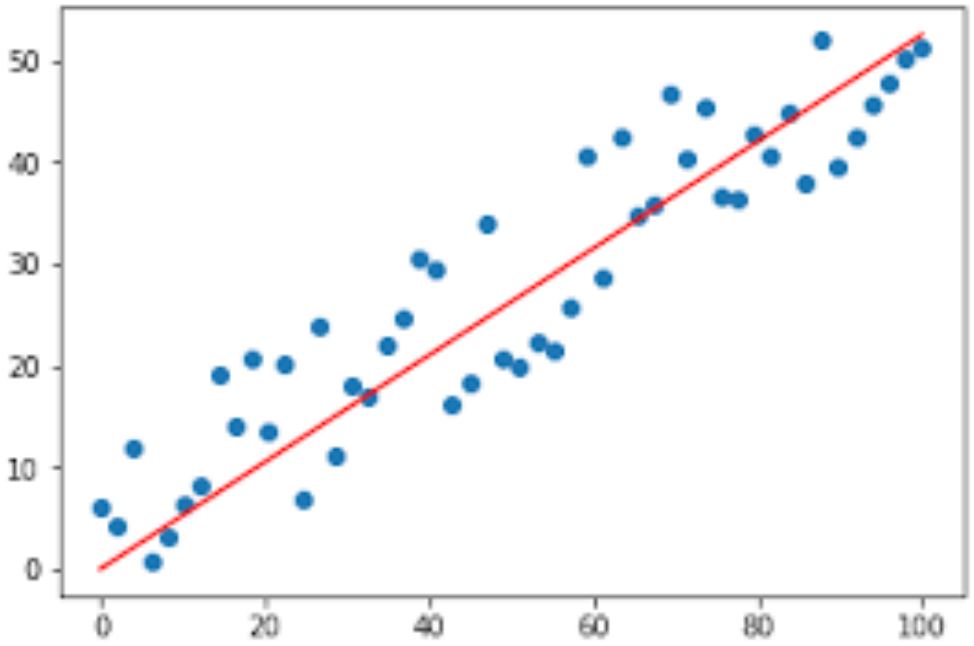
Project One – Part 1

Python tutorial: http://learnpython.org/

TensorFlow tutorial: https://www.tensorflow.org/tutorials/

PyTorch tutorial: https://pytorch.org/tutorials/

Linear Regression



$$\hat{y} = w[0] \times x[0] + w[1] \times x[1] + \ldots + w[n] \times x[n] + b$$

Linear Regression

$$\hat{y} = w[0] \times x[0] + w[1] \times x[1] + \ldots + w[n] \times x[n] + b$$

Simple linear regression (mean squared error):

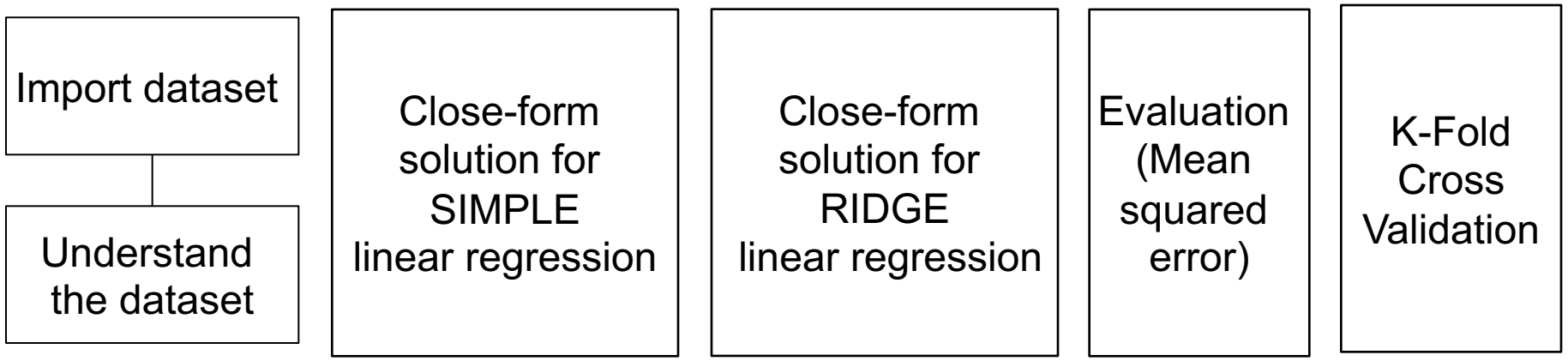
$$\sum_{i=1}^{M} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{M} \left(y_i - \sum_{j=0}^{p} w_j \times x_{ij} \right)^2$$

Ridge linear regression (mean squared error + regularization term):

$$\sum_{i=1}^{M} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{M} \left(y_i - \sum_{j=0}^{p} w_j \times x_{ij} \right)^2 + \lambda \sum_{j=0}^{p} w_j^2$$

regularization term

Main Modules for Part 1



Main Modules for Part 1

Import dataset

Understand the dataset

Importing the dataset

In [ ]:
```
# Import the boston dataset from sklearn
# Load dataset to some variable
# boston_data = .....
```

In [2]:
```
#  Create X and Y variables - X holding the .data and Y holding .target
# X = boston_data.....
# y = boston_data.....

#  Reshape Y to be a rank 2 matrix using y.reshape()

# Observe the number of features and the number of labels
# print('The number of features is: ', X.shape[1])
# Printing out the features
# print('The features: ', boston_data.feature_names)
# The number of examples
# print('The number of exampels in our dataset: ', X.shape[0])
# Observing the first 2 rows of the data
# print(X[0:2])
```

```
The number of features is:  13
The features:  ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
The number of exampels in our dataset:  506
[[6.3200e-03 1.8000e+01 2.3100e+00 0.0000e+00 5.3800e-01 6.5750e+00
  6.5200e+01 4.0900e+00 1.0000e+00 2.9600e+02 1.5300e+01 3.9690e+02
  4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 0.0000e+00 4.6900e-01 6.4210e+00
  7.8900e+01 4.9671e+00 2.0000e+00 2.4200e+02 1.7800e+01 3.9690e+02
  9.1400e+00]]
```

Main Modules for Part 1

Close-form solution for SIMPLE linear regression

```
In [6]: # Define the get_coeff_ridge_normaleq function. Use the normal equation method.
        # Return w values

        def get_coeff_linear_normaleq(X_train, y_train):
            # use np.linalg.pinv(...)
            return w
```

Closed-form solution:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$$

Main Modules for Part 1

Close-form solution for RIDGE linear regression

```
In [5]: # Define the get_coeff_ridge_normaleq function. Use the normal equation method.
        # Return w values

        def get_coeff_ridge_normaleq(X_train, y_train, alpha):
            # use np.linalg.pinv(...)
            return w
```

Closed-form solution:

$$W = (X^T X + \lambda I)^{-1} X^T y$$

$$I: identity\ matrix$$

Main Modules for Part 1

Evaluation (Mean squared error)

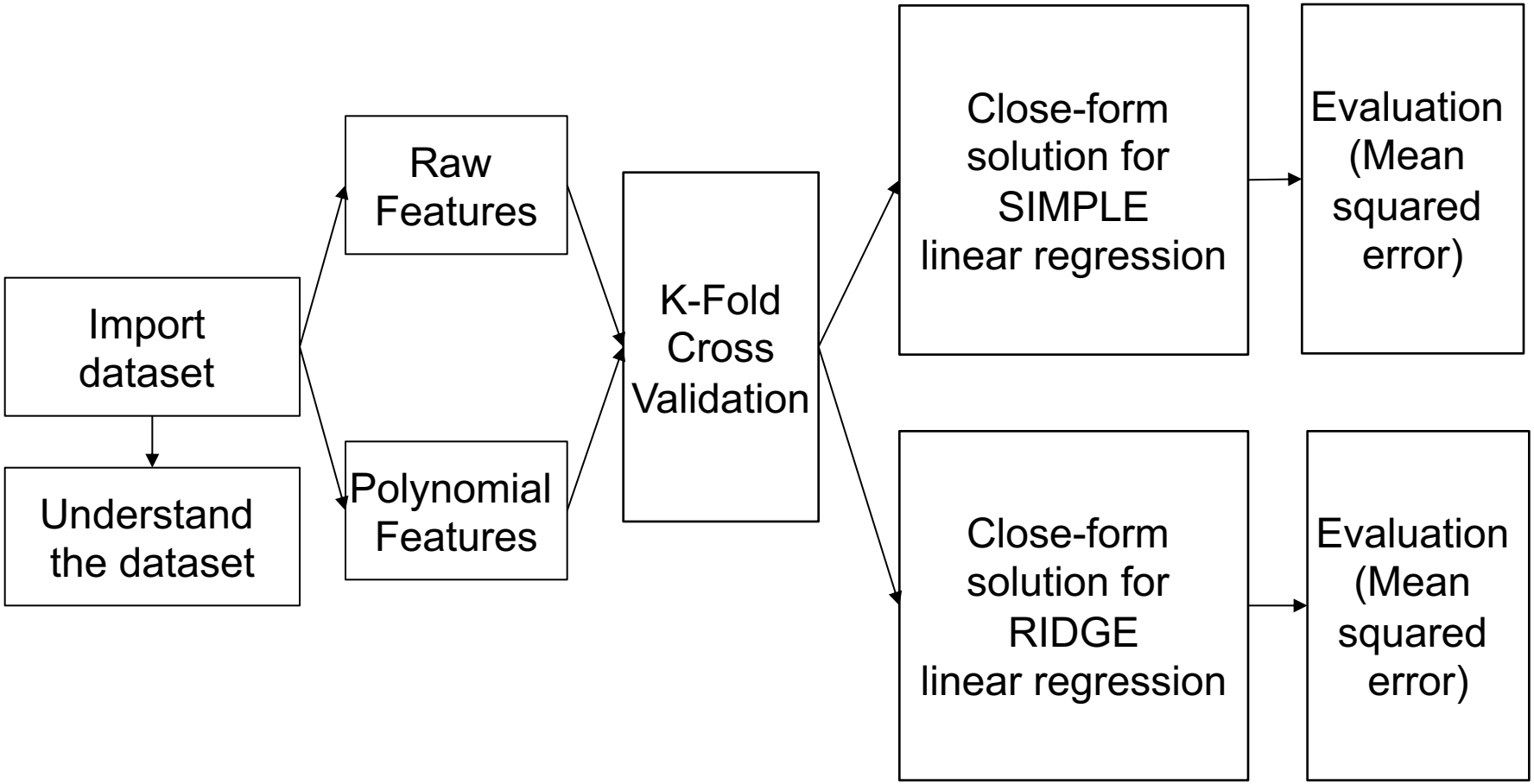
In [7]:
```
# Define the evaluate_err_ridge function.
# Return the train_error and test_error values


def evaluate_err(X_train, X_test, y_train, y_test, w):
#     pred_train=prediction using w and X_tran+np.mean(y_train)
#     pred_test=prediction using w and X_test
#     remember to add the mean back
#     train_error=...
#     test_error=...

    return train_error, test_error
```

Main Modules for Part 1

K-Fold Cross Validation

In [8]:
```
# Finish writting the k_fold_cross_validation function.
# Returns the average training error and average test error from the k-fold cross validation
# Sklearns K-Folds cross-validator: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFo

def k_fold_cross_validation(k, X, y, alpha):
    kf = KFold(n_splits=k, random_state=21, shuffle=True)
    total_E_val_test = 0
    total_E_val_train = 0
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Centering the data so we do not need the intercept term (we could have also chose w_0=average y value)
        # Subtract y_train_mean from y_train and y_test
        # y_train_mean = ...
        # y_train = ...
        # y_test = ...

        # Scaling the data matrix
        # Using scaler=preprocessing.StandardScaler().fit(...)
        # And scaler.transform(...)
        # X_train =
        # X_test =

        # Determine the training error and the test error
        # Use get_coeff_linear_normaleq or get_coeff_ridge_normaleq to get w
        # And use evaluate_err()

       ##############
    return  total_E_val_test, total_E_val_train
```

Pipeline for Part 1



Average Errors for Different λ

```
-------------------- 10.0 --------------------
Test Error 0.05201045114741994 Train Error 0.0480740034202244
-------------------- 31.622776601683793 --------------------
Test Error 0.05272915867716548 Train Error 0.048935977285238255
-------------------- 100.0 --------------------
Test Error 0.05551965535350536 Train Error 0.05209812996241122
-------------------- 316.22776601683796 --------------------
Test Error 0.06464019252599232 Train Error 0.06185014075217414
-------------------- 1000.0 --------------------
Test Error 0.08664578892409816 Train Error 0.08461165728446099
-------------------- 3162.2776601683795 --------------------
Test Error 0.11991925843943999 Train Error 0.11859250439076517
-------------------- 10000.0 --------------------
Test Error 0.1530153811412051 Train Error 0.15208559884333156
-------------------- 31622.776601683792 --------------------
Test Error 0.17322757905384642 Train Error 0.17244343969322065
-------------------- 100000.0 --------------------
Test Error 0.18168511049803734 Train Error 0.18094696997126405
-------------------- 316227.7660168379 --------------------
Test Error 0.18463936439006468 Train Error 0.1839155662655611
-------------------- 1000000.0 --------------------
Test Error 0.18560467517979254 Train Error 0.18488538284721184
-------------------- 3162277.6601683795 --------------------
Test Error 0.18591315202109848 Train Error 0.18519528124141094
-------------------- 10000000.0 --------------------
Test Error 0.18601102641875986 Train Error 0.18529360483210366
-------------------- Linear --------------------
Test Error 0.051897062420884064 Train Error 0.04788358273133742
```

Machine Learning, Spring 2020

Project One – Part 2

Python tutorial: http://learnpython.org/

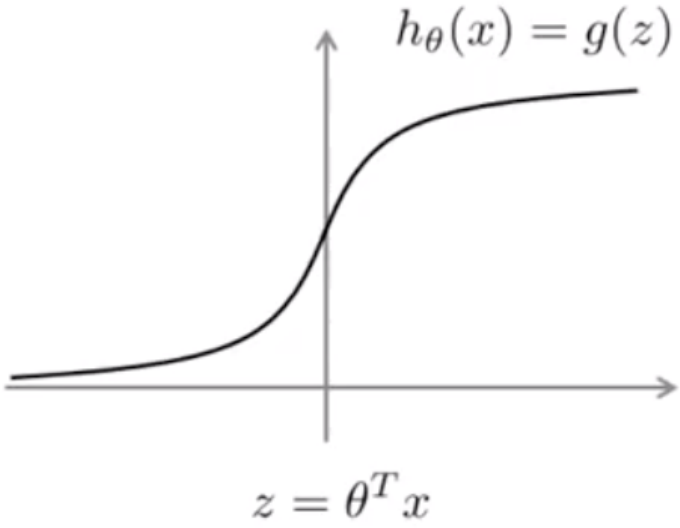
TensorFlow tutorial: https://www.tensorflow.org/tutorials/

PyTorch tutorial: https://pytorch.org/tutorials/

Logistic Regression

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

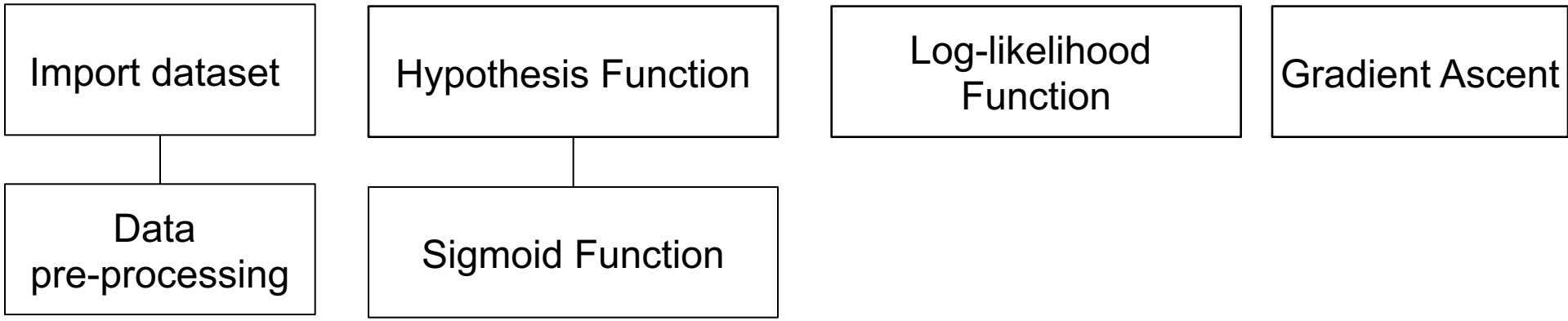
hypothesis function



If y =1, we wish our predicted hypothesis value is close to 1, then $\theta^T x \gg 0$

If y =0, we wish our predicted hypothesis value is close to 1, then $\theta^T x \ll 0$

Main Modules for Part 2



Main Modules for Part 2

Import dataset

Importing the dataset

In [1]:
```
# Load dataset to a python variable cancer
# Store target to a variable called y
# Store feature to a variable called X
```

In [2]:
```
# Printing the shape of data (X) and target (Y) values
# print(X.shape)
# print(y.shape)
```

Understanding the dataset

In [70]:
```
# Printing the names of all the features
print(cancer.feature_names)
```

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Main Modules for Part 2

Data pre-processing

Data Pre-Processing

Splitting the data into train and test before scaling the dataset

In [66]:
```
# Use train_test_split() function to split the dataset
# Store the return value of pervious step to X_train, X_test, y_train, y_test
```

Scale the data since we will be using gradient ascent

In [84]:
```
# Find the scaler of the dataset by using preprocessing.StandardScaler().fit()
# Using this scale to scale the X_train and X_test using .transform()
```

In [68]:
```
# TODO - Print the shape of x_train and y_train
# print(X_train.shape) # It should print (426, 30)
# print(y_train.shape) # It should print (426,)
```

Main Modules for Part 2

Data pre-processing

Adding a column of ones to the matrices X_{train} and X_{test}

After adding a column of ones $X_{train} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_1^{(N')} & x_2^{(N')} & \dots & x_d^{(N')} \end{bmatrix}$

Similarly for X_{test}

In [3]:
```
# Append a column of ones to x_train

# Create a column vector of ones by using np.ones and reshape
# Append a column of ones in the beginning of x_train by using np.hstack

# Now do the same for the test data

# We can check that everything worked correctly by:
# Printing out the new dimensions
print("The trainng data has dimensions: ", X_train.shape, ". The testing data has dimensions: ",X_test.shape)
# Looking at the first two rows of X_train to check everything worked as expected
print(X_train[0:2])
```

```
('The trainng data has dimensions: ', (426, 31), '. The testing data has dimensions: ', (143, 31))
[[1.000e+00 9.668e+00 1.810e+01 6.106e+01 2.863e+02 8.311e-02 5.428e-02
  1.479e-02 5.769e-03 1.680e-01 6.412e-02 3.416e-01 1.312e+00 2.275e+00
  2.098e+01 1.098e-02 1.257e-02 1.031e-02 3.934e-03 2.693e-02 2.979e-03
  1.115e+01 2.462e+01 7.111e+01 3.802e+02 1.388e-01 1.255e-01 6.409e-02
  2.500e-02 3.057e-01 7.875e-02]
 [1.000e+00 1.195e+01 1.496e+01 7.723e+01 4.267e+02 1.158e-01 1.206e-01
  1.171e-02 1.787e-02 2.459e-01 6.581e-02 3.610e-01 1.050e+00 2.455e+00
  2.665e+01 5.800e-03 2.417e-02 7.816e-03 1.052e-02 2.734e-02 3.114e-03
  1.281e+01 1.772e+01 8.309e+01 4.962e+02 1.293e-01 1.885e-01 3.122e-02
  4.766e-02 3.124e-01 7.590e-02]]
```

Main Modules for Part 2

Hypothesis Function

Sigmoid Function

Our hypothesis, $h(\mathbf{x})$

The next function to write is our hypothesis function.

For example if our design matrix X consists of single example $X = [1, x_1, x_2, \ldots, x_d]$ and weights $\mathbf{w}^T = [w_0, w_2, \ldots, w_d]$, it returns $h(\mathbf{x}) = \frac{1}{1+e^{-(w_0 \cdot 1 + w_1 \cdot x_1 + \cdots w_d \cdot x_d)}}$

If given a matrix consisting of N' examples $X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \ldots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \ldots & x_d^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_1^{(N')} & x_2^{(N')} & \ldots & x_d^{(N')} \end{bmatrix}$ and weights $\mathbf{w}^T = [w_0, w_2, \ldots, w_d]$, the function returns a column vector $[h(\mathbf{x}^{(1)}), h(\mathbf{x}^{(2)}, \ldots, h(\mathbf{x}^{(N')}]^T$

In [75]:
```
# Predict the probability that a patient has cancer
# Write the hypothesis function
def hypothesis(X , w):

    return
```

Sigmoid(z)

The first function you will write is sigmoid(z)

sigmoid(z) takes as input a column vector of real numbers, $z^T = [z_1, z_2, \ldots, z_{N'}]$, where N' is the number of examples

It should produce as output a column vector $\left[\frac{1}{1+e^{-z_1}}, \frac{1}{1+e^{-z_2}}, \ldots, \frac{1}{1+e^{-z_{N'}}}\right]^T$

In [6]:
```
# Write the sigmoid function
def sigmoid(z):

    return
```

Main Modules for Part 2

Log-likelihood Function

Log-Likelihood Function.

Write the code to calculate the log likelihood function $\ell(\mathbf{w}) = \sum_{i=1}^{N'} y^{(i)} \ln(h(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \ln(1 - h(\mathbf{x}^{(i)}))$

The input is a matrix consisting of N' examples $X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_1^{(N')} & x_2^{(N')} & \dots & x_d^{(N')} \end{bmatrix}$ and a column vector $\mathbf{y}^T = [y^{(1)}, y^{(2)}, \dots, y^{(N')}]$ of labels for X.

The output is $\ell(\mathbf{w})$

In [85]:
```
# Write the log likelihood function
def log_likelihood(X , y , w ):

    return
```

Main Modules for Part 2

Gradient Ascent

Gradient Ascent

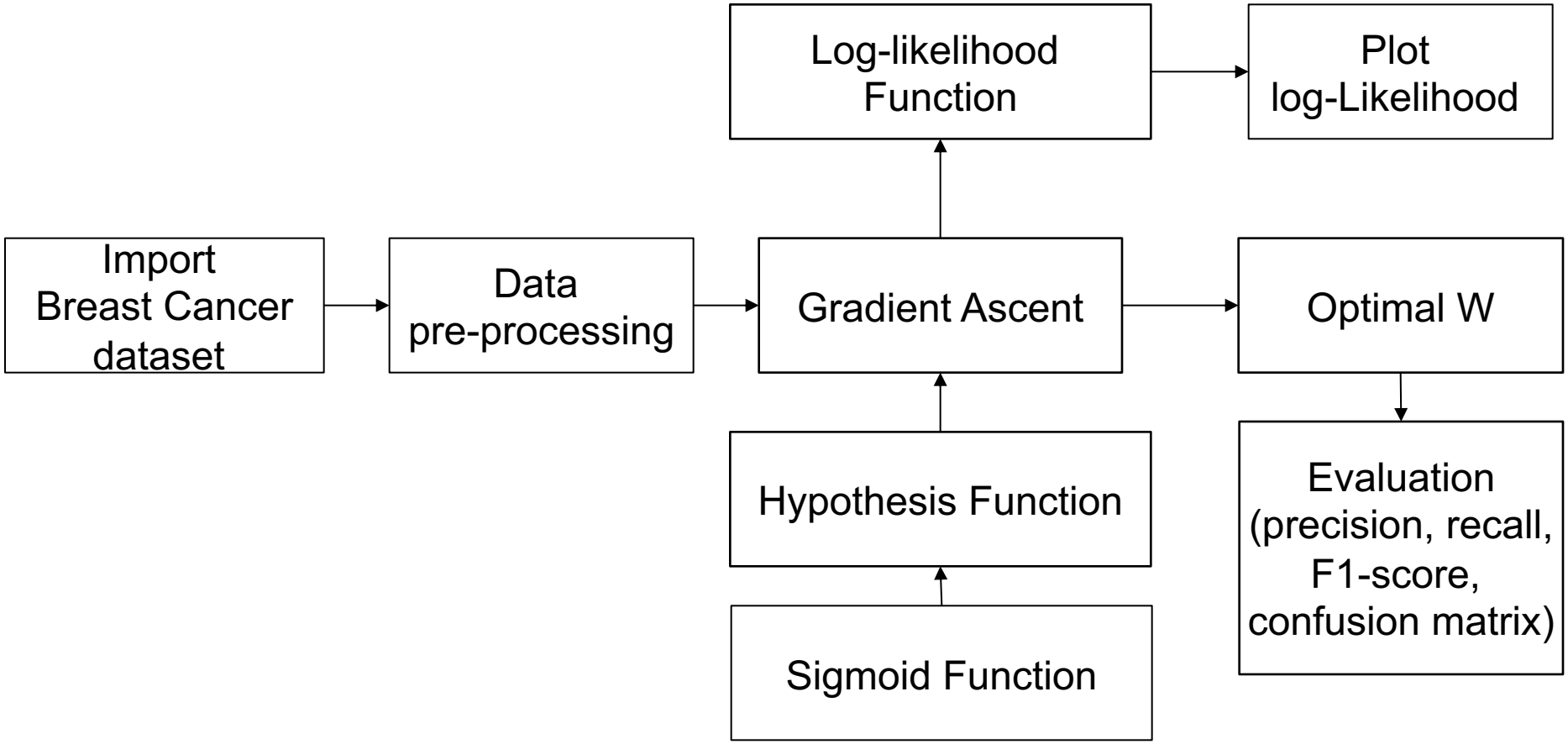
Now write the code to perform gradient ascent. You will use the update rule from the lecture notes.

In [89]:
```
# TODO - Write the gradient ascent function
def Logistic_Regresion_Gradient_Ascent(X, y, learning_rate, num_iters):
    # For every 100 iterations, store the log_likelihood for the current w
    # Initializing log_likelihood to be an empty list
    # Initialize w to be a zero vector of shape x_train.shape[1],1
    # Initialize N to the number of training examples
    for i in range(num_iters):
        # update the w using formula
        # append the log_likelihodd values to the list for every 100 iterations

    return w, log_likelihood_values
```

In [90]:
```
learning_rate = 0.5
num_iters = 5000 # The number of iteratins to run the gradient ascent algorithm

w, log_likelihood_values = Logistic_Regresion_Gradient_Ascent(X_train, y_train, learning_rate, num_iters)
print(w)
# print(log_likelihood_values)
```

```
[[ 0.        ]
 [-0.26280797]
 [ 0.21403595]
 [-0.14545374]
 [-0.44812872]
 [ 0.15519867]
 [ 2.64830449]
 [-1.3770735 ]
 [-2.75456517]
 [ 0.9948169 ]
 [ 0.12207317]
 [-3.30847177]
 [ 0.19895472]
 [-0.70978165]
 [-2.03234292]
 [ 0.86954988]
 [ 0.47404246]
 [ 1.44431141]
 [-1.90479258]
```

Pipeline for Part 2

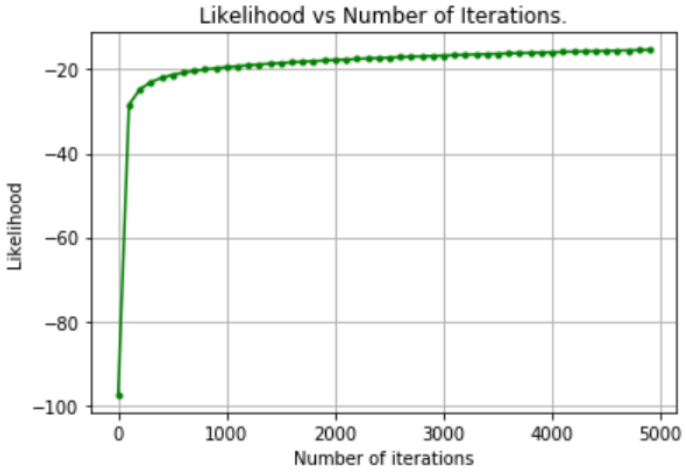


Pipeline for Part 2

Plot log-Likelihood

Plotting Likelihood v/s Number of Iterations.

In [91]:
```
# Run this cell to plot Likelihood v/s Number of Iterations.
iters = np.array(range(0,num_iters,100))
plt.plot(iters,log_likelihood_values,'.-',color='green')
plt.xlabel('Number of iterations')
plt.ylabel('Likelihood')
plt.title("Likelihood vs Number of Iterations.")
plt.grid()
```



Evaluation

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

- **Confusion Matrix**
  - TP: A **true positive** is an outcome where the model *correctly* predicts the *positive* class.
  - TN: Similarly, a **true negative** is an outcome where the model *correctly* predicts the *negative* class.
  - FP: A **false positive** is an outcome where the model *incorrectly* predicts the *positive* class.
  - FN: And a **false negative** is an outcome where the model *incorrectly* predicts the *negative* class.

Evaluation

- **Precision**
  - TP / (TP + FP)
  - The % of correctly predicted sample among all the samples predicted 1

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

For example,

- 50 samples have cancer (actual 1)
- 50 samples do not have cancer (actual 0)
- 45 samples predicted 1 (40 TP + 5 FP)
- 55 samples predicted 0 (45 TN +10 FN)

- → Precision: 40 / (40 + 5) = 88.89%

Evaluation

- **Recall**
  - TP / (TP + FN)
  - The % of correctly predicted sample among all the samples actual 1

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

For example,

- 50 samples have cancer (actual 1)
- 50 samples do not have cancer (actual 0)
- 45 samples predicted 1 (40 TP + 5 FP)
- 55 samples predicted 0 (45 TN +10 FN)

→ Precision: 40 / (40 + 5) = 88.89%
→ Recall: 40 / (40 + 10) = 80%

Evaluation

- **F1-score**
  - 2*(Precision * Recall) / (Precision + Recall)
  - A measure that combines precision and recall

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

For example,

- 50 samples have cancer (actual 1)
- 50 samples do not have cancer (actual 0)
- 45 samples predicted 1 (40 TP + 5 FP)
- 55 samples predicted 0 (45 TN +10 FN)

→ Precision: 40 / (40 + 5) = 88.89%
→ Recall: 40 / (40 + 10) = 80%
→ F1- score: 2*(88.89% * 80%) / (88.89% + 80%) = 84.21%

References

- Scikit-learn documentation: http://scikit-learn.org/stable/modules/svm.html
- https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative

Up next: a short break, and then Python examples for support vector machines.