refers to the linear form of the model

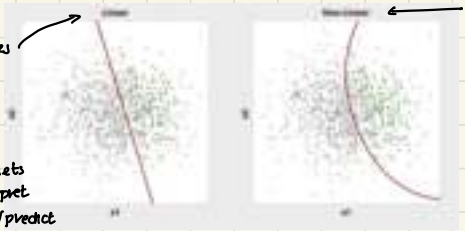
PRELAB #4

- Our focus will be on a method called LOGISTIC REGRESSION.
- We'll learn a method called GRADIENT DESCENT
  - ↳ A general purpose numerical optimization technique you use when trying to find model weight that minimizes a loss function during training.
- Linear models are a class of supervised learning models that are represented by an equation and use a linear combination of features and weights to compute the label of an unlabeled example.

LINEAR REGRESSION: linear model in supervised learning used in classification problems to predict the probability of an outcome. To optimize the log loss.

- These models can be represented by a single equation.
- MODEL TRAINING involves:
  - Initializing weights in the equation with a random guess
  - Slowly adjusting the weights in the principled way to improve the model's prediction performance.
- LINEAR MODELS: supervised learning models that use the linear combination of features & weights to compute the label of an unlabeled example.
  - ↳ attempt to improve themselves during training by using an optimization technique that minimizes training loss.
  - ↳ optimization technique uses a loss function to quantify the amount of error the model makes against the training dataset.
- Starts with LOSS FUNCTION
- The LOSS FUNCTION guides the process for adjusting the model's weights.
  - ↳ NOTE: in kNN and DT, loss function is used to guide the training process.

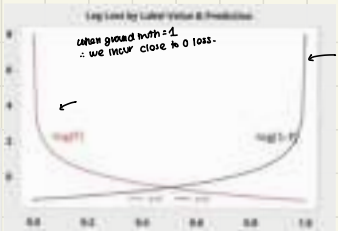
function that maps the features to the output is a straight line of the form $y=ax+b$



- can take on many forms
- like a straight line but bending it into a more suitable shape.
- nonlinear has more complexity. Meaning they can draw more sophisticated curves.
- can fit on data points that a linear cannot.

Advantages:
- ↳ less complex
- ↳ Good with small data sets
- ↳ Easier to explain/interpret
- ↳ Fast to train/implement/predict

- Neural networks generally fit data better, but with their increased complexity, they can lead to massive overfitting and poor generalization in certain cases. ∴ it's important to embrace simpler techniques in logistic regression.
- DECISION DESIGN: choosing a linear vs. nonlinear design
- LOSS FUNCTION: mathematical way of representing a model's prediction error.
  - ↳ Helps us quantify if our predictions are correct by how correct it is.
  - ↳ Each loss function takes in 2 inputs: predicted value + ground truth label
  - ↳ Different types of loss function:
    1. LOG LOSS: used for training classification models
       - ↳ used for both NEURAL NETWORK & LOGISTIC REGRESSION.
       - ↳ Compares prediction probabilities to ground truth ∴ it allows us to measure different shades of correctness.
       - ↳ Many loss functions are just for evaluation because they aren't suitable for numeric optimization techniques, which is a core requirement for loss functions when doing model training.
       - ↳ focusing on loss function to specifically guide the training algorithm to update model weights. We'll focus on LOG LOSS & MEAN SQUARED ERROR.



LOSS FUNCTION

↳ Used to evaluate a model on training data & tells us how bad its performance is.
↳ Determines how far the predicted labels from the actual labels in the training data.
↳ Higher loss = worse
loss = 0 means the model makes perfect predictions.
↳ TRAINING PROCESS:
- Involves optimization
- Linear model uses high loss to determine its performance.
- Inaccurate predictions = HIGH loss value
  Accurate predictions = LOW loss value

↳ EXAMPLES OF LOSS FUNCTIONS: Common to divide the overall training loss by the total number of training examples N, so that the output can be the average loss per example.

NOTE: Loss function = cost function

Ex) STEP 1: initialize the model

① LOG LOSS (AKA. BINARY CROSS ENTROPY LOSS)

Commonly used to measure the performance of a binary classification model S.A. logistic regression.

$$L_{LL} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1-y_i)\log(1-p_i)\right)$$

y_i: label of a training example
p_i: probability prediction value for the same training example

② MEAN SQUARE ERROR (MSE)

Used to measure the performance of a regression model S.A. linear regression.

For every example, you take the difference between the label & prediction & square it.

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

y_i: label of training example
$\hat{y}_i$: prediction of training example

③ ZERO-ONE LOSS

Used in classification to evaluate classifiers in multiclass/ binary classification settings but is rarely useful to guide optimization during training.

Counts how many mistakes a model makes when making a prediction.
For every single example that's predicted incorrectly, suffers a loss of 1.
The normalized zero-one loss returns the fraction of misclassified training examples also referred to as the training error.

$$L_{0/1} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(\hat{y}_i \neq y_i), \text{ where } \mathbb{1}(\hat{y}_i \neq y_i) = \begin{cases} 1, & \text{if } \hat{y}_i \neq y_i \\ 0, & \text{otherwise} \end{cases}$$

LOGISTIC REGRESSION

↳ Use it when low model complexity is a desired feature. (simplicity)
Ex) making predictions on medical data; collect data sets from patient data & for them to be really small and also the need for interpretability to be very high.

↳ STEP 1: Prep our data
STEP 2: Train the model

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(C=1)
model.fit(X_train, y_train)
probs = model.predict_proba(X_test)
```

logistic regression is specifically designed to return probabilities.

NOTE: typically when you have small data and/or a lot of features you'll probably use a LOWER C VALUE

Main hyperparameter = C: controls the complexity
higher model complexity = less regularization = higher C
lower model complexity = more regularization = lower C
↳ NOTE: We don't know which value is the best ∴ test different values.

	Feature	Weight
0	revenue	0.001870
1	outcalls	-0.000061
2	incalls	-0.002711
3	months	-0.010282
4	eqpdays	0.001127
5	webcap	-0.268256
6	marryyes	0.003588
7	travel	-0.030808
8	pcown	0.033175
9	creditcd	-0.067064
10	retcalls	0.624961

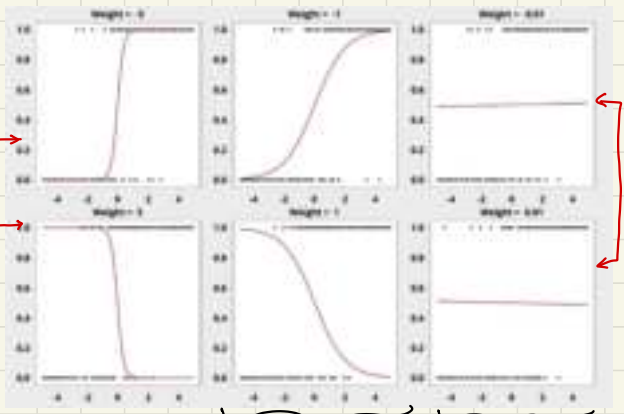
Represented by letter B (beta)
Intercepted by letter a (alpha)
In ML it's represented by W

- The weights and intercepts are stored in the model object.

```
def log_reg_predict(X, weight, alpha):
    '''
    X is array; weight is array; alpha is float
    '''
    #compute linear input
    MX = alpha + (X * weight).sum()

    #use inverse logit to get Prob(Y|X)
    P = 1 / (1 + np.exp(-1*MX))

    return P
```

intercept
← compute linear component of prediction

Illustrative visualization of processes behind logistic regression: Plotting a numeric feature against a binary outcome



- RED curves are the output to the inverse logic
- Linear portion of the model (or $X \cdot W + Alpha$) can take the values of $[-\infty, \infty]$
  - Slope of S shape curve = determined by the feature weight.
  - Direction of curve = sign of weight

y values cluster at opposite ends of the X values

Y values do cluster but the pattern is harder to decipher

MODERATE WEIGHT
↳ Slope is more gradual

Feature has NO predictive value ∴ the weight will be 0 or ~0

Slopes are so small, cannot see the S curve unless we extend X axis to $-\infty$

USING LOGISTIC REGRESSION TO MAKE PREDICTIONS

- Best suited for binary classification
- It can also be used for multi-class classification by making minor tweaks to the algorithm.
- Belongs to a class called GENERAL LINEAR MODEL that uses the linear combination of features and sets of weights to compute the label of an unlabeled example.
- When using logistic regression model to make class predictions we take the feature values of an unlabeled example, pass them through our logistic regression model, which performs 3 steps to output a label:
  1. LINEAR STEP: Generating an output by taking the sum of feature values times their learned weights.
  2. INVERSE LOGIT (SIGMOID): Using an inverse logit function to transform the output of STEP 1 into a probability prediction btw 0-1
  3. MAPPING (THRESHOLDS): Using probability threshold to output the class label from the probability prediction generated in STEP 2

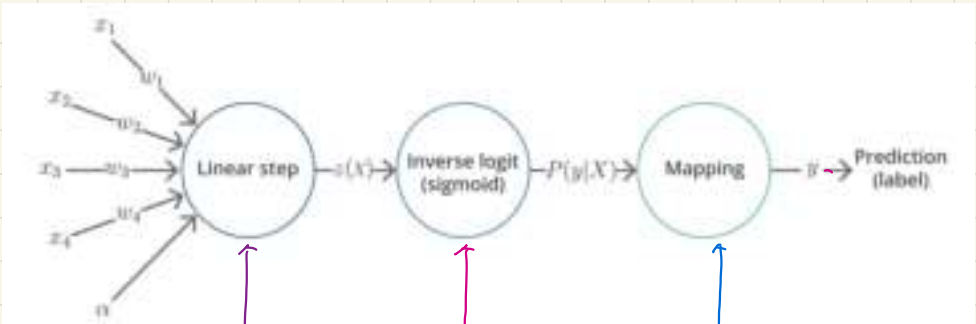


Figure 1. Anatomy of a Logistic Regression model for binary classification.

Computes a value z(X) by:

$$z(X) = a + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

intercept (Alpha) — linear sum of features value X model weights W

NOTE: if examples only have 1 feature; the linear step of the logistic regression model is basically the function of a line.

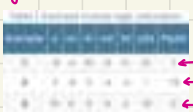
- Weights & intercept = MODEL PARAMETERS
  part of the model & are learned during the training phase from the training data.

  tells us how an individual feature x affects the likelihood of the example belonging to a certain class.

Transforms the output of linear step into probability prediction P(y|X) between 0-1:

$$P(y|X) = \frac{1}{1+e^{-z(X)}}$$

- Large (-) # = closer to 0 = inverse logit step is 0.5
- Large (+) # = closer to 1
- Plot various values of z(X) vs P(y|X) you'll end up with the following:



← high confidence it IS a spam
← Equal chance of not being spam
← high confidence it's NOT a spam

We can map the probability values generated by a logistic regression model to binary class labels

Ex) spam vs. no spam:
map probability close label 0 to represent not spam
map probability class label 1 to represent spam

We can do this by choosing a threshold then use a conditional statement with the threshold to map the probability to a class label.

$$y = \begin{cases} 1 & \text{if } P(y|X) > \text{threshold} \\ 0 & \text{else} \end{cases}$$

IN CONCLUSION...

These 3 steps make up the DECISION BOUNDARY

EXERCISE: LOGISTIC REGRESSION

Use logistic regression model & analyze its performance.

Our problem is a BINARY CLASSIFICATION problem. We're trying to predict whether a customer will leave their current telecom company or whether a customer will not leave the company. This problem is well suited for a LOGISTIC REGRESSION MODEL.

STEP 1: LOAD A "READY TO FIT" DATA SET

```
import pandas as pd
import numpy as np
import os

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
from sklearn.metrics import accuracy_score

filename = os.path.join(os.getcwd(), "data", "cell2cell_train.csv")
df = pd.read_csv(filename, header=0)
```

STEP 2: CREATE LABELED EXAMPLES FROM THE DATA SET

A. IDENTIFY NUMERIC COLUMNS TO USE AS FEATURES

To implement a Logistic Regression model, we can ONLY use the numeric columns.

Use Pandas DataFrame select_dtypes() to obtain all of the column names that have a dtype of float64. Save to a list called feature_list

```
feature_list = list(df.select_dtypes(include="float64"))
```

B. CREATE X AND y LABELED EXAMPLES

Get the Churn column from df and assign it to y. This will be our LABEL which will either be T/F

Get the columns listed in feature_list from df and assign them to X. This will be our FEATURES.

```
y = df["Churn"]
X = df[feature_list]
print("Number of examples: " + str(X.shape[0]))  #510447
print("Number of features: " + str(X.shape[1]))  #35
```

STEP 3: CREATE TRAINING AND TEST DATA SETS

1. Use scikit-learn train_test_split() to create the data sets.
2. Specify:
   - ↳ A test set is 33% of the size of the data set
   - ↳ Seed value of 1234

```
X_test, X_train, y_test, y_train = train_test_split(X,y, test_size = 0.33, random_state = 1234)
```

Check the dimensions of the training and test datasets are what you expect:

```
print(X_train.shape)  # (15315, 81)
print(y_train.shape)  # (35732, 81)
```

STEP 4: FIT A LOGISTIC REGRESSION CLASSIFICATION MODEL & EVALUATE THE MODEL

Complete training the logistic regression classification model, analyze its performance and print it out.

Train a logistic regression model on the training data, test the resulting model on the test data, and compute and return:

(1) The log loss of the resulting probability predictions on the test data
(2) The accuracy score of the resulting predicted class labels on the test data.

NOTE: evaluating a model's training loss & evaluating a model's accuracy is different.

1. Use LogisticRegression() to create a model object & assign the result to model. You'll not provide arguments for its hyperparameter but use scikit-learn's default value.
2. Call model.fit() to fit the model to the training data. The first argument should be X_train and the 2nd is y_train
3. Call the model.predict_proba() with argument X_test to use the fitted model to predict values for the test data. Store it to probability_predictions.

   NOTE: predict_proba() returns 2 columns: 1 column per class label.
   The first column contains the probability that an unlabeled example belongs to class False (Churn is "False")
   The 2nd column contains the probability that an unlabeled example belongs to class True (Churn is "True")
4. Call the log_loss() the first argument is y_test and the 2nd argument is probability_predictions. Assign result to l_loss

   RECALL: loss indicates how close the prediction probability is the actual class label.
   the closer the probability is to the label (ex. probability of 0.9 that the class label of "False" when the actual class label is indeed False), the lower the loss.
5. Call the model.predict() with the argument X_test to use the fitted model to predict the class labels for the test data. Store the outcome to class_label_predictions

   NOTE: predict() returns the class label (T/F) per unlabeled examples.
6. Call the accuracy_score(): the first argument should be y_test and the 2nd argument should be class_label_predictions. Assign the result to acc_score

   NOTE: Accuracy refers to the number of class label predictions that are correct.

```
model = LogisticRegression()
model.fit(X_train, y_train)
probability_predictions = model.predict_proba(X_test)

#print the first 5 probability class predictions
df_print = pd.DataFrame(probability_predictions, columns=['Class: False', 'Class: True'])
print("Class Prediction Probabilities: \n" + df_print[0:5].to_string(index=False))

l_loss = log_loss(y_test, probability_predictions)
print('Log loss: ' + str(l_loss))

class_label_predictions = model.predict(X_test)
print("Class labels: " + str(class_label_predictions[0:5]))

acc_score = accuracy_score(y_test, class_label_predictions)
print("Accuracy: " + str(acc_score))
```

STEP 5: THRESHOLDS: MAP PROBABILITIES TO A CLASS LABEL

Note that `predict_proba()` returns 2 columns:

1st column: contains the probability that an unlabeled example belongs to a class `False`
2nd column: contains the probability that an unlabeled example belongs to a class `True`

To determine the ORDER OF THE CLASSES in `predict_proba()` output, you can inspect the model's object's `classes_` property.

```
print (model. classes_ ) # [False  True]
```

NOTICE: the probabilities map to labels in the table below. The table contains:

- 3 unlabeled examples
- The resulting class probability values from the logistic regression model's `predict_proba()`
- The corresponding class label from the same logistic regression model's `predict()`

Inspect "Example 1":

	Class: False	Class: True	Class Label
Example 1	0.745386	0.2546614	False
Example 2	0.745386	0.2546614	False
Example 3	0.4360333	0.5639667	True

The `predict()` assigns Example 1 the class label False.

For binary classification, the method defaults to a 0.5 threshold.

↳ resulting probability for class 0 is >= 0.5 ∴ The unlabeled example is given a label of False
↳ resulting probability for class 0 is <= 0.5 ∴ The unlabeled example is given a label of True

Sometimes we may want a different threshold.

The function `computeAccuracy()` takes a threshold value as an argument that does the following:

1. Loops through the array `probability_predictions` (obtained from the Logistic Regression model)
   - It extracts the first column's probability
   - It checks if that probability is greater than or equal to the threshold value
   - If so, it assigns a class label of `False`. Otherwise it assigns a class label of `True`
   - It saves the new class label to list `labels`.
2. Compute the accuracy score by comparing the new class labels contained in list `labels` with the ground truth labels contained in `y_test`
3. Returns the accuracy score

```
def computeAccuracy (threshold_value):
    labels = []
    for p in probability_predictions[:, 0]:
        if p >= threshold_value:
            labels. append (False)
        else:
            labels. append (True)
    acc_score = accuracy_score (y_test, labels)
    return acc_score
```

EXERCISE COMPLETE

Call the `computeAccuracy()` with few different threshold values.

```
thresholds = [0.44 , 0.50, 0.55, 0.69, 0.75]
for t in thresholds :
    print("Threshold value {:. 2f}: Accuracy {3." format (t, str (computeAccuracy (t))))
```

IMPLEMENT THE INVERSE LOGIT AND LOG LOSS

Demonstrate how to perform vector and matrix operations using Numpy to achieve higher efficiency.
Do this by implementing inverse logit function to make predictions & loss function to evaluate those predictions.

INTRODUCTION TO MATHEMATICAL COMPUTATION

Linear algebra concepts involve operations on VECTORS & MATRICES.

A lot of data manipulation in pandas relies on these concepts.

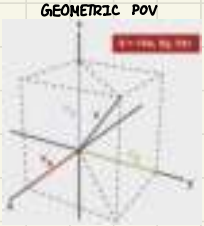
VECTOR OPERATIONS WITH NUMPY

Mathematical operations we apply to Numpy array and the 3 mathematical operations are:

1. ELEMENT-WISE ARITHMETIC
2. SUMMATION
3. DOT PRODUCT

- These 3 operations are sufficient for building a predict function for computing the log loss.
- In Python we use the array datatype from NumPy. In traditional math we call them ARRAY VECTORS.
- GOAL: Transform math into code
- VECTOR: set of numbers that usually represent coordinates in a geometric space

GEOMETRIC POV



- 3 dimensions
- Represented by 3 points

VECTORS WE DEAL WITH: DATA

UnitID	X_1	X_2	X_3
AAA111	0.47	0.01	0.28
ABF123	0.52	0.15	0.70
AAO666	0.09	0.57	0.64
ABC134	0.07	0.58	0.41
AIH222	0.63	0.81	0.98
AAB248	0.00	0.73	0.33

- Think of SINGLE DATA COLUMN/ROW as a VECTOR

CONTEXT IN WHICH WE'RE GOING TO USE THEM: INVERSE LOGIT

$$P(y|X) = (1+e^{-(\alpha+XW)})^{-1}$$

- X: vector of feature values
- α: the intercept
- W: weight we learned in the logistic regression

- There's 1 weight associated with each feature used to model
- DOT PRODUCT:

$$X = < x_1, x_2, \dots x_n >$$
$$W = < w_1, w_2, \dots, w_n >$$

Both are vectors of size N

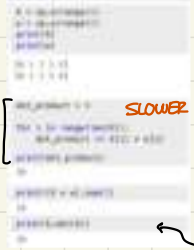
In python these would be Numpy arrays

Dot Product:

$$X \cdot W = \sum_{i=1}^{n} x_i * w_i$$
$$= x_1 w_1 + x_2 w_2 + \dots + x_n w_n$$

you can think of SIGMA COMMAND as a for loop where we're incrementing the sum as we iterate over the particular sequence

3 CODING APPROACHES:



APPROACH 1

implementing the sum as a loop and adding the product of array elements to a running sum

HOWEVER... general rule in Numpy is to avoid explicitly writing loops as much as possible.

SLOWER

$= x_1 w_1 + x_2 w_2 + \dots + x_n w_n$

APPROACH 2

Taking the element wise product, taking advantage of Numpy's vectorization functionality & the summing using the sum method

$= \sum_{i=1}^{n} x_i \cdot w_i$

APPROACH 3

Takes the dot product of 2 vectors. $= X \cdot W$

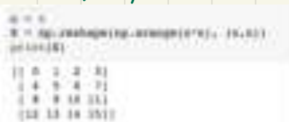
ONLY WORKS WHEN THE 2 VECTORS ARE THE SAME SIZE

MATRIX OPERATIONS

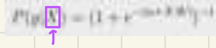
Many of the equations that define ML methods use matricies in their notation

MATRIX: multidimensional array

EX) 4x4 NumPy array



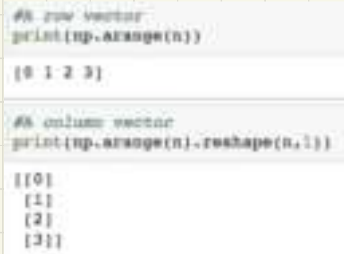
EX) INVERSE LOGIT



vector or matrix
Treat X as a matrix which corresponds to making predictions for all examples in the data at the same time.

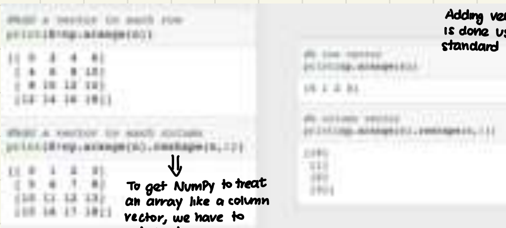
TO SELECT ROWS, COLUMNS, AND VALUES:

IMPLEMENTATION & OUTPUT:



- Ensure that the dimensions are the same for each matrix

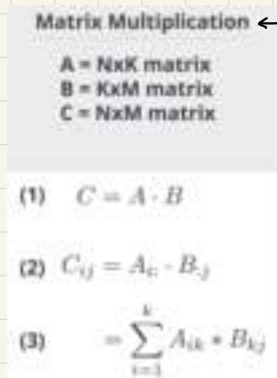
① ADDITION: ADDING VECTOR TO MATRIX



Adding vectors to the matrix is done using the same standard addition operator

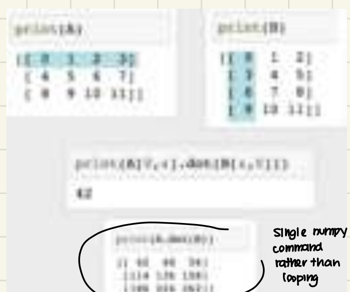
To get NumPy to treat an array like a column vector, we have to reshape it.

③ MATRIX MULTIPLICATION



When multiplying matricies to be the same size

What matters is that the inner dimensions of the matricies are the same.

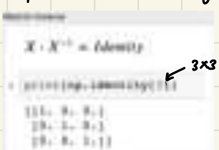


Single numpy command rather than looping

EX)

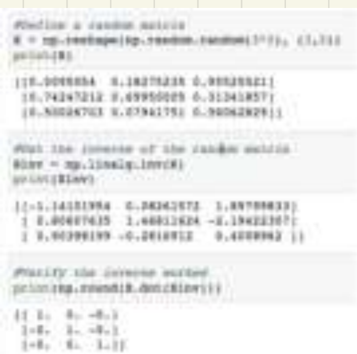
④ **INVERSE OF A MATRIX:** a matrix that when multiplied by the og matrix then it produces an identity matrix.

- **IDENTITY MATRIX:** special matrix whose diagonal elements are 1 and all other are 0.
- **MATRIX INVERSE:**



3x3 matrix

- **MATRIX INVERSE IN NUMPY:**



LOGISTIC REGRESSION: PREDICTING FROM SCRATCH

We're going to train a logistic regression model then implement our own version of scikit-learn predict_proba function to make predictions.

Ex) 11 features and we're predicting customer churn which is represented in the last column.

- STEP 1: run import, split data, and train the model

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = df[['creditscore', 'geography', 'gender', 'age', 'tenure',
        'balance', 'numofprod', 'hascrcard', 'isactivemember',
        'estimatedsalary']]
y = df['exited']

X_train, X_test, y_train, y_test = train_test_split(X, y)

logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

- STEP 2: implement the inverse logit from scratch & make predictions
  - Equivalent to using scikit-learn's built in prediction_proba()

$$P(y|X) = (1 + e^{-(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p)})^{-1}$$

① Since X is a matrix, we'll treat X as a matrix

```
X = X_test.to_numpy()
weight = logreg.coef_
print(X.shape)
print(weight.shape)
```

FIRST CONVERT DATAFRAME TO A MATRIX
PULL WEIGHTS FROM THE LOGISTIC REGRESSION
CHECK DIMENSIONS OF OUR DATA MATRIX

```
(7972, 11)
(1, 11)
```

← They don't align well ∴ TRANSPOSE

```
XW = X.dot(weight.T)
print(XW.shape)
print(XW[0:5])
```

← TRANSPOSE
← now same row & column as input matrix

```
(7972, 1)
[[ 0.2188888 ]
 [-0.60567796]
 [ 0.1288894 ]
 [-0.81112472]
 [ 0.2481677 ]]
```

TRANSPOSE: Reverses the shape
↳ use it if the dimensions don't align well.
↳ Ex) (11, 1) ⟹ T ⟹ (1, 11)

```
XW = X.dot(weight.T).ravel()
print(XW.shape)
print(XW[0:5])
```

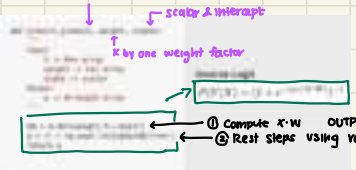
```
(7972,)
[ 0.2188888 -0.60567796  0.1288894 -0.81112472  0.2481677 ]
```

↳ ENCAPSULATE IT INTO A FUNCTION THAT COMPUTES THE FULL INVERSE LOGIT:

data matrix: contains shape & examples & features

scalar & intercept

K by one weight factor



① Compute x·w OUTPUT: array of length N
② Rest steps using vectorization

with inputs:

```
True
```

EXERCISE: IMPLEMENTING THE PREDICTION STEP & EVALUATING THE MODEL

We'll be practing how to implement the DOT PRODUCT and taking the INVERSE of a matrix. This will be used to build a fnchon that implements the LOGISTIC REGRESSION PREDICTION STEP, and to build a function that implements LOG LOSS.
We will then compare LOSS fnchon with scikit-learn's implementation.

IMPORT THE PACKAGES:

```
import pandas as pd
import numpy as np
from sklearn.metrics import log_loss
```

STEP 1: MATRIX MULTIPLICATION & INVERSE

Start with 100 x 3 matrix. We'll use a multivariate random number generator to create synthetic data.

```
np.random.seed(1234)  #seed the generator
mean_vector = [0, 0, 0]
cov_matrix = [[1, 0.25, 0.25],
              [0.25, 1, 0.25],
              [0.25, 0.25, 1]]
X= np.random.multivariate_normal(mean_vector, cov_matrix , size = 3)
```

Use np.cov() to compute the covariance matrix of X.
Use the parameter rowvar= False when calling the np.cov(). This makes sure that we treat the columns as the features and ensures that we return 3x3 matrix.

```
cov_numpy = np.cov(X, rowvar= False)
```

Now we'll compute the same covariance matrix manually. The steps are the following:

1. Using the NumPy mean() to compute the mean of each column in array X.
   - REMEMBER: use axis=0 to make sure we'll compute the mean of each column.
   - Assign this to X_means
2. We then need to subtract the mean from each column of NumPy array X.
   - Create a data matrix called X_centered which is just the OG NumPy array X with X_mean subtracted from it.

```
X_means = X.mean(axis=0)
X_centered = X - X_means
```

Now that we have a CENTERED DATA MATRIX, we need to compute the COVARIANCE MATRIX.

COVARIANCE MATRIX: $\frac{X_c^T \cdot X_c}{N-1}$ ← centered matrix

centered matrix TRANSPOSED

REMEMBER: The inner dimension of the 2 matrix needs to align. If it isn't aligned we need to TRANSPOSE

Implement the covariance matrix formula using the X_centered
Remember to use Numpy T attribute to get a transpose and the Numpy dot() to compute the dot product.
Assign the result to cov_manual

```
cov_manual = X_centered.T.dot(X_centered)/(X.shape[0]-1)  # Equivalent to the dot product of X_centered transpose & X_centered
```

Now let's see if our computation of covariance matrix matches the Numpy version.
We'll do one preprocessing step. We may not expect the lower decimals to equal each other ∴ we round each covariance matrix to desired tolerance & check the equality of those rounded matricies.

1. Create a new matrix called cov_numpy_round using the NumPy np.round_() fnchon. Use the cov_numpy matrix for the 1st argument and use the tolerance variable for the 2nd argument (the decimal input)
2. Create a new matrix cov_manual_round using the NumPy np.round_() fnchon. Use the cov_manual matrix for the 1st argument and use the tolerance variable for the 2nd argument (the decimal input)
3. Now test the equality of the two rounded covariance matrices:
   A. Write a boolean expression to test equality between cov_numpy_round & cov_manual_round
   B. Apply the sum() on the expression to see if it's equal to 9. This should return either True / False
   C. Assign variable to result

```
tolerance = 10

cov_numpy_round = np.round_(cov_numpy, decimals = tolerance)
cov_manual_round = np.round_(cov_manual, decimals = tolerance)

result = ((cov_numpy_round == cov_manual_round).sum() == 9)  #True
```

Now we'll call the inverse of the covariance matrix. The inverse of the covariance matrix is used to compute a distance metric called: MAHALANOBIS DISTANCE.
MAHALANOBIS DISTANCE is used to indicate that computing an inverse covariance matrix has utility in certain ML applications.

1. Use the Numpy np.linalg.inv() to compute the inverse of cov_manual. Assign it to cov_inv
2. Use the Numpy dot() to get the dot product of cov_manual and cov_inv to see if it returns a 3x3 identity matrix. Assign the result to cov_dot and print the variable.

```
from numpy.linalg import inv  # import numpy.linalg.inv

cov_inv = inv(cov_manual)
cov_dot = np.dot(cov_manual, cov_inv)
```

STEP 2: IMPLEMENT THE PREDICTION STEP: PREDICTING PROBABILITIES

We want to implement the prediction step of a logistic regression model and then evaluate the model using our own implementation of the log loss function.

$$\text{LogLoss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i * \log(p_i) + (1-y_i) * \log(1-p_i))$$

n = total number of training examples
y_i = label for a given example
p_i = Logistic Regression prediction which is determined by the following inverse logit function: $p_i = \frac{1}{(1+e^{-(x_i \cdot w + a)})}$

x_i = feature vector of training examples
W = logistic regression weight vector
a = intercept

Let's begin with a function that'll implement the prediction step.

- compute probabilities
- implementing the inverse logit

1. Compute the linear portion $x \cdot W + a$. Assign the result to xw
2. Compute the probabilities $\frac{1}{(1+e^{-xw})}$. Assign the result to p

```
import math
def compute_lr_prob(X, w, alpha):
    # X = NxK data matrix (array)
    # w = Kx1 weight vector (array)
    # alpha = scalar intercept (float)
    xw = X.dot(W).ravel() + alpha
    p = 1/(1 + np.exp(-1 * xw))
    return p

W = [1, -1, 2]
alpha = -1
p = (compute_lr_prob(X, w, alpha))
```

STEP 3: IMPLEMENT LOG LOSS

Implement the log loss equation: $\text{LogLoss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i * \log(p_i) + (1-y_i) * \log(1-p_i))$. Store this to log_loss

```
def compute_log_loss(y, p):
    n = len(y)
    divide_n = 1/n
    negative_divide_n = -1 * divide_n
    sum_log_loss = np.sum(y * np.log(p) + (1-y) * np.log(1-p))
    log_loss = negative_divide_n * sum_log_loss

    return log_loss
```

STEP 4: USE LOG LOSS TO EVALUATE THE MODEL'S PREDICTION

Now we'll test our compute_log_loss() function
First we generate some labels y to compare against our predictions.

```
W = [1, -1, 2]
alpha = -1
y = (compute_lr_prob(X, W, alpha) > np.random.rand(X.shape[0])).astype(int)
```

Now evaluate our predictions using our log loss implementation.

1. Create an array of probabilities called p using compute_lr_prob() using X, W, and alpha
2. Compute log loss using compute_log_loss() and using the p array and the y array of ground truth labels. Store in log_loss_manual

```
p = compute_lr_prob(X, w, alpha)
log_loss_manual = compute_log_loss(y, p)
log_loss_manual  # 0.3387821864064603
```

STEP 6: COMPARE OUR LOG LOSS IMPLEMENTATION WITH SCIKIT-LEARN'S

Compute log loss using scikit-learn and the same `p, y` arrays as our log loss function above. Store the resulting log loss in `log_loss_sklearn`. Compare the resulting log loss with our own implementation.

```
log_loss_sklearn = log_loss(y, p)
print(log_loss_sklearn)  # 0.33782216404603
print(log_loss_manual == log_loss_sklearn)  # True
```

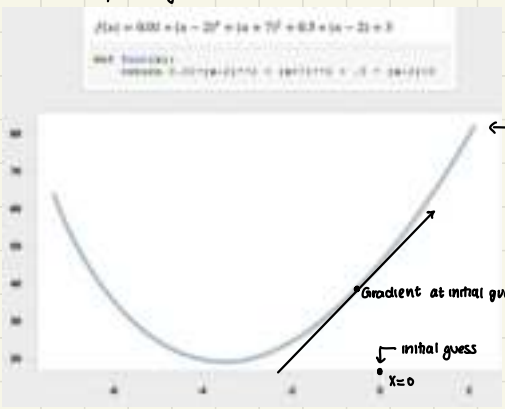
EXERCISE COMPLETE

TRAIN A LOGISTIC MODEL REGRESSION

- Employs **GRADIENT DESCENT** which is an **OPTIMIZATION ALGORITHM** used to find the **minimum of a function**. Used to **iteratively train** and optimize a **logistic regression function**.

GRADIENT DESCENT: numeric optimization algorithm used to find the minimum and maximum of arbitrary mathematical functions.

↳ In ML it's used to find **optimal weights** of logistic regression.
↳ It does this by **minimizing a loss function**. It finds the weights that results in the lowest training loss.
↳ GOAL: find the value of x that results in the lowest value of $f(x)$
↳ 3 different ways of doing this:



← GRAPH:
① Guess on where the minimum lies
↳ common starting point is $x=0$
② Then compute the gradient of the function at the value of the initial guess.
↳ Gradient is the **first derivative** of the function you're trying to minimize.
↳ Gradient is the **SLOPE at a given point**.
↳ Gradient arrow **UPWARDS** = **positive** value
Gradient arrow **DOWNWARDS** = **negative** value
↳ Gradient guides us by showing us whether the function is increasing or decreasing when we move to a specific direction.
- At our INITIAL GUESS we want to INCREASE the function b/c it's +.
- But since we want to **MINIMIZE** the function, we need to move in the **opposite direction of the gradient**.

③ Now consider HOW FAR should we move the gradient.

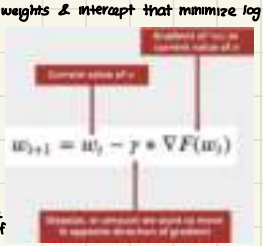
↳ If the gradient is close to 0 then we can stop the procedure.
- Up to us to define the stopping criteria

↳ Think of x = **FEATURE WEIGHT**
$f(x)$ = **LOG LOSS**

↳ **IMPLEMENTING GRADIENT DESCENT**
- for logistic regression function would be log loss
- Apply gradient descent to find model weights & intercept that minimize log loss.
- General gradient descent algorithm:

GOAL: search over different values of w to find the one that minimizes the function

Gradient descent is an iterative process:
1. Start with initial guess
2. Compute gradient of the guess
3. Update our guess by moving some amount in the opposite direction of the gradient

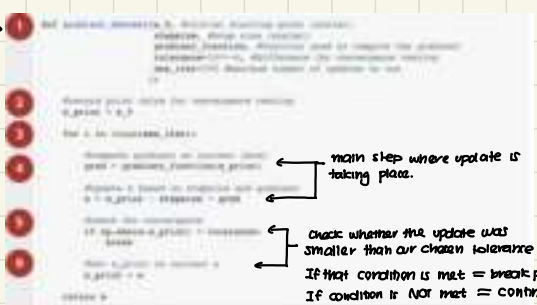


Ex) gradient descent

Horizontal line has a slope of 0
When slope = 0 ∴ no direction → when this condition is met we've found our **MINIMUM** ★★★★

- IN CODE:

Pass in initial guess b/c we need to see process with the starting point. You can use any value.



- We need to modify the function to operate on a vector of weights. We'll apply a special technique to getting **LEARNING RATES**.

HOW TO CHOOSE A LEARNING RATE

In the GRADIENT DESCENT algorithm we have to choose an APPROPRIATE LEARNING RATE.

- Learning rate too big / too small ═══ iterative gradient CONVERGES
- Learning rate impacts how long it'll take for the algorithm to converge.
- OBJECTIVE: find values of w that minimizes some function F(w)

current value of w ↓ — gradient of F(w) at current value of w ↓

$$W_{t+1} = W_t - H(w)^{-1} * \nabla F(W)$$

↑ 2nd derivative of F(w) (Hessian) at current value of w

1st derivative ═ slope of the function
2nd derivative ═ helps us understand the curvature of our function

- Updated code using Hessian to help us find learning rates:

helps us takeout the guess work in here

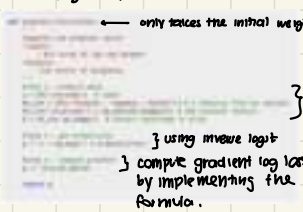
This method mathematically yields the fastests and most accurate convension speeds.

USING GRADIENT DESCENT IN LOGISTIC REGRESSION TRAINING

Use gradient descent function and make a few modifications.

STEP 1: run gradient descent on full vector of weights as opposed to one weight
STEP 2: implement our gradient & hessian function

- Gradient of negative loss: $\nabla LL = -1 * (Y - P) \cdot X$
- Computing gradient:
  - STEP 1: Compute model predictions for each example represented by vector P.
  - STEP 2: Take difference between ground truth labels Y and the predictions P & take the dot product of this vector with the original feature matrix X.



← only takes the initial weight vector as input

} How we get gradient descent method to learn the intercept

} using inverse logit
} compute gradient log loss by implementing the formula.

Intercept: weight for a feature that has constant value of one for all examples.

Gets the same treatment as the features

- Implement the HESSIAN FUNCTION
  - The Math: $Q = P * (1 - P)$

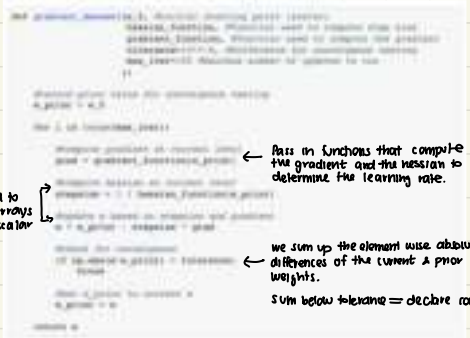
$$\text{Hessian} = (X * Q)^T \cdot X$$

} Result is a $k \times k$ matrix where K is the number of features plus 1 for the intercept.

} prepare data to compute model predictions

NOW PUTTING IT TOGETHER...

full gradient descent function:



adapted to work on arrays instead of scalar values

← pass in functions that compute the gradient and the hessian to determine the learning rate.

← we sum up the element wise absolute differences of the current & prior weights.

sum below tolerance = declare convergence

ex)

- we've using gradient descent to learn the intercept.

To test whether the gradient descent provided the right weights in the first place.

To do this compare our implementation with an implementation we trust.

- We call our custom function and also a fit model using scikit-learn.
- second learn uses regularization by default.
- Gradient descent is the method of choice for training advanced neural networks using applications like image classifications, recommendation engines, and natural language processing.

TRAINING A LOGISTIC REGRESSION MODEL

The training process of logistic regression model is accomplished iteratively with the help of LOSS FUNCTION & OPTIMIZATION ALGORITHM.

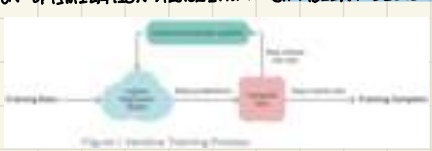
uses the LOSS FUNCTION to evaluate a model's loss and then adjusts the model parameters accordingly to reduce loss.

It continues until an optimal model is produced.

GOAL: Finding the parameters (weights & intercept) that'll result in the best model.

COMMON OPTIMIZATION ALGORITHM: GRADIENT DESCENT

↳



During the model instantiate, the weights and intercept are usually set to some small random float numbers.

The model will typically start off performing poorly since it was created at random.

As it progresses iteratively throughout the training phase, its weight and intercept become more & more optimized against the training data set and the performance improves.

LOSS FUNCTION: Evaluates the performance of the model against training data at any point in time.

↳ model misclassifies = high loss = weights + intercept will be tweaked for optimization
↳ model classifies correctly = low loss = weights + intercept will no longer be updated

↳ MOST COMMON LOSS FUNCTION: **LOG LOSS/CROSS ENTROPY**

$$L(x_i, y_i) = -y_i * \log(p_i) - (1-y_i) * \log(1-p_i)$$

- Misclassified = log loss will yield a high value

Ex) The logistic regression's inverse logit function outputs a prediction probability of 0.014 that the example belongs to class label 1.
∴ our model has incorrectly predicted the label for this training example.

Log loss: $L(x_i, y_i) = -1 \log(0.014) - 1 - 1 \log(1-0.014) = 1.82$ ← HIGH LOG LOSS b/c model was UNSUCCESSFUL at predicting the label

- classified correctly = log loss will yield a low value

Ex) The logistic regression's inverse logit function outputs a prediction probability of 0.96 that the example belongs to class 1.
0.96 is high probability ∴ model predicted correctly for this label.

Log loss: $L(x_i, y_i) = -1 \log(0.96) - 1 - 1 \log(1-0.96) = 0.0177$ ← LOW LOG LOSS b/c model was successful at correctly predicting the label.

↳ Actual training includes many training examples.

For each iteration we would evaluate the loss of our model against a set of training examples. This allows us to get a snapshot of the performance of our logistic regression model with the weights and the intercept at that particular time.

The LOSS for the batch examples is simply the sum of their individual losses by the number of examples.

AVERAGE LOSS at batch level is referred as **COST FUNCTION**.

↳ Examples misclassified = loss of this batch of examples is HIGH
Examples classified correctly = loss of this batch of examples is LOW

$$L(\bar{x}, \bar{y}) = -\frac{1}{n} \sum_{i=1}^{n} (y_i * \log(p_i) + (1-y_i) * \log(1-p_i))$$

GRADIENT DESCENT: updates weights and the intercepts on poorly performing models. It's the optimization algorithm.

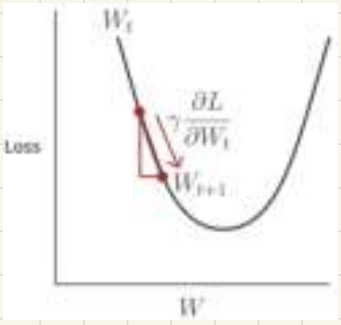
↳ Start with an arbitrary value for model parameters. It can be a random value but in most cases it's 0.
↳ We take the partial derivative ⇓ of the loss function L with respect to the weights $\bar{w}$ and the intercept a individually
⇓
Tells us how much & towards what direction (positive/negative) to update the weights & intercepts.

↳ We then update the model parameters accordingly.
- The learning rate γ determines the **STEP SIZE / speed** at which we update the model parameters.
  ↳ Typical γ ranges between 0.001 and 0.1
- SMALL LEARNING RATE = making small changes to the model parameter.
  HIGH LEARNING RATE = everytime when updating the model parameter we're taking a big step.

↳ FORMULA to compute the next value of weights and the intercept at given update step:

$$\bar{w}_{t+1} = \bar{w}_t - \gamma \frac{\partial L}{\partial w_t} \quad a_{t+1} = a_t - \gamma \frac{\partial L}{\partial a_t}$$

↳ How gradient does to adjust a weight during each update step:



- curve represents all possible values of w for that one weight
- GRADIENT of LOSS is equal to the PARTIAL DERIVATIVE (slope) of the curve and tells you the DIRECTION and HOW MUCH TO MOVE towards the BOTTOM of the curve.
- Log loss function is CONVEX (∴ only 1 MINIMUM)
- GOAL: iteratively update the value of W to reduce loss until we reach some global minimum. (until slope is ~0)

∴ identified value of W will result in the lowest training loss.

PSEUDOCODE FOR TRAINING A LOGISTIC REGRESSION MODEL:

1. Initialize a random numbers $\overline{w}$ equaling the number of features x.
2. Choose a learning rate γ
3. For a subset of training examples, calculate the loss $L(x, y)$
4. Update $\overline{w}$ and a using gradient descent:

$$\overline{w}_{t+1} = \overline{w}_t - \gamma \frac{\partial L}{\partial \overline{w}_t}$$

$$a_{t+1} = a_t - \gamma \frac{\partial L}{\partial a_t}$$

5. Repeat until the change in loss is smaller than some threshold.

LOGISTIC REGRESSION HYPERPARAMETER: LEARNING RATE

Logistic regression have their own hyperparameters.

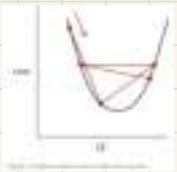
↳ By optimizing these hyperparameter, we can produce a model that generalizes well.

↳ common hyperparameter: LEARING RATE

LEARING RATE (γ): Also known as STEP SIZE that dictates the speed of gradient descent.

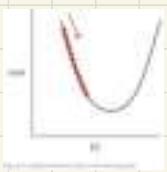
① HIGH LEARNING RATE

- Updates the weights and intercepts in big increments.
- With high learning rate the weights can fail to minimize & it'll just bounce back & forth near the bottom & it'll never get close enough to the minima



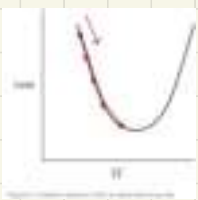
② LOW LEARNING RATE

- Updates the weights and intercepts in small increments.
- It'll take a long time to train the model since the weights and intercept change in every small increments during training



③ IDEAL LEARNING RATE

- Allows fast convergence to the minima without overshooting
- Choosing the right learning rate ensures efficient training of the model, reducing cost & improving model accuracy.



REGULARIZATION

A model can have low training loss but not generalize well to new data. This is OVERFITTING and is caused by a complex model that has learned the training data so closely that it does not generalize well to new data.

GOAL: minimize loss plus avoid overfitting by minimizing the model's complexity ⟹ REGULARIZATION helps

REGULARIZATION:

↳ Penalizes complex models in an attempt to prevent overfitting.

↳ By adding regularization terms which are functions of the weights, we can mitigate overfitting by "punishing" a model with extreme values for any given weight.

↳ 2 ways to think of model complexity:

① model complexity is a function of the TOTAL NUMBER OF FEATURES with non-zero weights.

② model complexity is a function of the WEIGHTS of all features in the model

↳ 2 TYPES OF REGULARIZATION used to address these 2 types of model complexity:

① L_1 regularization (LASSO)

Represents the loss as computed by the loss function

L1 introduces a term that penalizes less important features, reducing their coefficients to 0 and effectively eliminating them.

L1 penalizes the count of non-zero weights.
L1 penalizes |weight|

② L_2 regularization (RIDGE)

L2 introduces a regularization term to the loss function that's the sum of squares of all feature weights.

L2 penalizes weight^2

- Prevents weights from growing out of porportion
- L_1 and L_2 adds a penalty to the loss function for weights that have grown too large.

Oftentimes our input feature values can be too large or too imbalanced. Due to this, these features can overly influence the size of their corresponding weights.

CHEAT SHEET

Logistic Regression

Algorithm Name	Logistic regression
Description	Logistic regression is a probabilistic linear model. In essence, a logistic regression classifier produces the probability $P(X\|y)$ via the inverse logit (sigmoid) function.
Applicability	Binary classification problems.
Assumptions	None.
Underlying Mathematical Principles	• Linear classification model • Inverse logit function • Log loss function
Additional Details	• We can use gradient descent to find the optimal solution. You can tune the learning rate. • Logistic regression uses regularization to reduce model complexity, and you can tune hyperparameter C to adjust the penalty.
Example	Predict whether a candidate will win an election based on features such as campaign funds, poll results, if the candidate is currently in office or not, etc.

EXERCISE: OPTIMIZING LOGISTIC REGRESSION

```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
% matplotlib inline


from sklearn.metrics import log_loss
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
```

Build a logistic regression models using "Cell2Cell" a telecom company churn prediction data set
You can build many variants each with different value of C hyperparameter which governs the amount of regularization used.

REGULARIZATION = process where we add penalty to the OG log loss function.

L2 REGULARIZATION:

$$\text{Regularized log loss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i * \log(p_i) + (1-y_i) * \log(1-p_i)) + \frac{1}{C}\sum_{j=1}^{m} w_j^2$$

- Penalty is the sum of the squares of the weights scaled by a constant 1/C
- Hyperparameter C is large == reduce weight of penalty == less regularization

We'll build a Logistic regressions with different values of C & analyze how it impacts the log loss.

In this assignment we'll follow the following:

1. Load "Cell2cell train" data set
2. Create unlabeled examples containing numerical features only
3. Split the data into training and test data sets
4. Fit logistic regression classifier using scikit-learn & evaluate the log loss & accuracy of the predictions.
5. Fit multiple Logistic regression classifiers with different values of the regularization hyperparameter C & plot the resulting log loss and accuracy.

STEP 1: LOAD THE DATA SET

```
filename = os.path.join(os.getcwd(), "data", "cell2celltrain.csv")
df = pd.read_csv(filename, header=0)
```

STEP 2: CREATE LABELED EXAMPLES FROM THE DATA SETS

To implement a logistic regression model, we can only use numeric columns!

Create labeled examples from df
Get the Churn column from df and assign it to y. This will be our LABEL which will either be T/F
Get the columns listed in feature_list from df and assign them to X. This will be our FEATURES.

```
y = df["Churn"]
X = df[feature_list]
print("Number of examples: " + str(X.shape[0]))  # 51047
print("Number of features: " + str(X.shape[1]))  # 35
```

STEP 3: CREATE TRAINING & TEST DATA SETS

1. Use scikit-learn train_test_split() to create the data sets.
2. Specify:
   - ↳ A test set is 33% of the size of the data set
   - ↳ Seed value of 1234

```
X_test, X_train, y_test, y_train = train_test_split(X, y, test_size = 0.33, random_state = 1234)
```

Check the dimensions of the training and test datasets are what you expect:

```
print(X_train.shape)  # (34201, 35)
print(X_test.shape)   # (16846, 35)
```

STEP 4: FIT A LOGISTIC REGRESSION CLASSIFIER & EVALUATE THE MODEL

Complete training the logistic regression classification model, analyze its performance and print it out.

Train a logistic regression model on the training data, test the resulting model on the test data, and compute and return:

(1) The log loss of the resulting probability predictions on the test data
(2) The accuracy score of the resulting predicted class labels on the test data.

Inspect the function definition `train_test_LR(X_train, X_test, y_train, y_test, c=1)`. This function expects the test & train datasets as well as a value for hyperparameter C.

Use scikit-learn `LogisticRegression` class.
Use `LogisticRegression()` to create a model object & assign the result to the variable `model`. Provide arguments `C=c`

```
def train_test_DT(X_train,X_test, y_train, y_test, c=1):
    # create scikit-learn LogisticRegression model object & assign it to variable model
    model = LogisticRegression(C=c)

    # Fit model to the training data below
    model.fit(X_train, y_train)

    # Make predictions on the test data using predict_proba() & assign it to probability_predictions
    probability_predictions = model.predict_proba(X_test)

    # compute the log loss on probability_predictions and save it to l_loss
    l_loss = log_loss(y_test, probability_predictions)

    # make predictions on test data using predict() & assign it to class_label_predictions
    class_label_predictions = model.predict(X_test)

    # compute accuracy score on class_label_predictions and save it to acc_score
    acc_score = accuracy_score(y_test, class_label_predictions)

    return l_loss, acc_score
```

TRAIN A MODEL & ANALYZE RESULTS

```
loss, acc = train_test_LR(X_train, y_train, X_test, y_test)
print("Log loss" + str(loss))  # Log loss: 0.5878612157234193
print("Accuracy: " + str(acc)) # Accuracy: 0.7097828377918972
```

STEP 5: TRAIN ON DIFFERENT HYPERPARAMETER VALUES & ANALYZE THE RESULTS

Now we'll adjust the C Regularization hyperparameter to check its impact on log loss.

In scikit-learn's `LogisticRegression`, the parameter `C` = inverse of regularization strength
↳ smaller values = stronger regularization.

Train & evaluate a different `LogisticRegression` model for every value of C in list `cs`

1. Initialize empty list called `ll_cs`. This will store the log loss for every model
2. Initialize empty list called `acc_cs`. This will store the accuracy score for every model
3. Loop that iterates over `cs`
   ↳ Call on `train_test_LR()` with training & test data with current value of C
   ↳ The function `train_test_LR()` returns 2 items:
      A. Append 1st item to `ll_cs`
      B. Append 2nd item to `acc_cs`

```
ll_cs = []
acc_cs = []

for c in cs:
    call = train_test_LR(X_train, y_train, X_test, y_test, c)
    ll_cs.append(call[0])
    acc_cs.append(call[1])
```

Now visualize the results:

```
plt.figure(figsize=(15,5))

ax = sns.barplot(x=cs, y=ll_cs)
g = ax.set_xticklabels([f'10^{i}' for i in range(-10,10)])
ax.set_xlabel("Regularization Hyperparameter: C")
ax.set_ylabel("Log loss")
ax.set_ylim([0.5,0.62])
g=plt.title("Log loss test performance by regularization weight C")
```

PLOT ACCURACY: Plots the accuracy for every value of hyperparameter C:

```
fig = plt.figure()
ax = fig.add_subplot(111)
x = np.log10(cs)

sns.lineplot(x=x, y=acc_cs, marker='o')

plt.title("Accuracy test performance by regularization weight C")
plt.xlabel("Regularization Hyperparameter: C")
plt.ylabel("Accuracy")
plt.show()
```

EXERCISE COMPLETE

EX) An example of L2 regularization

Assume we have 4 features that means we have 4 weights.

At given point during training, the values of the 4 weights are: $\{w_1 = 3.1,\ w_2 = 5.2,\ w_3 = 1493,\ w_4 = -4.3\}$

Our loss function will return extremely high since w_3 has a very high value.

w_3 main cause of the model complexity as it's a much higher value than rest of the weights.

Determine the L2 regularization term that'll be added as penalty to the loss function:

$= w_1^2 + w_2^2 + w_3^2 + w_4^2$

$= 3.1^2 + 5.2^2 + 1493^2 + (-4.3)^2$

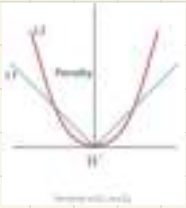
$= 9.61 + 27.04 + 2,229,049 + 18.49 = 2,229,104.14$

L2 regularization term that'll be added to the loss function as a penalty is 2,229,104.14. This encourages our model to reduce the model complexity

HYPERPARAMETER C: controls how much regularization is applied to the model.

The penalties are often accompanied by a factor C that controls the strength of the penalty.

Along with choosing the right regularization (L_1 and L_2), the value of C can be fine-tuned to improve the performance of the model.



→ As W becomes larger, the penalty will be higher for both L_1 and L_2
It tends to exert a downward pressure regardless of the magnitude of W

→ Even as the weights are getting closer to 0, it'll still exert pressure to attempt to bring it even closer to 0.

→ L2 reduces downward pressure as weights approach 0.
This is why L1 regularization is more conducive to a sparse model in which some of the weights end up having little or no effect on the model.

∴ L1 can be considered an act of feature selection, as it eliminates some weights of the features.

INTRODUCTION TO LINEAR REGRESSION

LINEAR REGRESSION model makes predictions based on the linear combination of model weights & features.

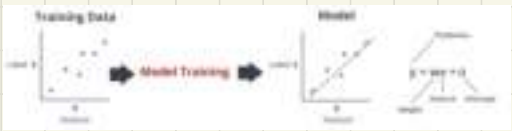
Difference between LINEAR REGRESSION and LOGISTIC REGRESSION:

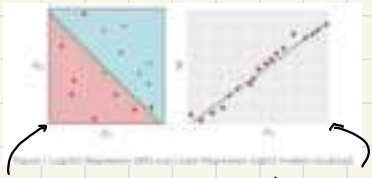
- Linear is used specifically for solving regression problems that have a continuous label S.A. age, temperature, distance, or salary.
- Logistic is a linear model that best suited to solve binary classification problems by predicting the probability of a class label S.A. probability that an email is spam
- Logistic is intended for classification problems.
- Linear is intended for regression problems.

- Logistic is used to predict the probability of a categorical label
- Linear is used to predict a continuous label S.A. price or age

LINEAR REGRESSION: makes predictions based on the linear combination of model weights and features.

↳ used to solve regression problems
↳ Finds linear relationship between one or more feature and a label
↳ Fits a linear model to the data by assuming that the data relationship is well described by a straight line
It uses that line to predict a label for a new unlabeled example.





Logistic regression is represented as a decision boundary between 2 classes

Linear regression is represented as a fitted line

MAKING PREDICTIONS WITH LINEAR REGRESSION:

- To make predictions we take the feature values of a new unlabeled example & input them into the fitted line (the equation)

MATHEMATICALLY: take the linear combination of the feature values $(x_1, x_2, \dots, x_n)$ and learned weights $(w_1, w_2, \dots, w_n)$ plus an intercept term (a). The result (y) is the label/prediction.

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + a$$

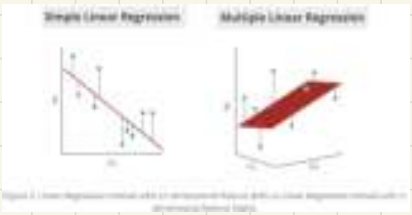
NOTE: intercept is commonly referred as the bias β.

- There are 2 types of linear regression models:

① SIMPLE LINEAR REGRESSION:
- Finds the linear relationship between one feature and one label
- Prediction for an example would be determined by using $y = w_1 x_1 + a$

NOTE:
feature = independent variable
label = dependent variable

② MULTIPLE LINEAR REGRESSION:
- Finds the linear relationship between multiple feature and one label
- Model will be a fitted hyperplane
- The prediction for an unlabeled example would be determined by $y = w_1 x_1 + w_2 x_2 + a$ for 2 FEATURES.



TRAINING A LINEAR REGRESSION MODEL:

- To train a linear regression model is to estimate the model parameters (weight and intercept) that best fit against the training data set.
- Methods of training:

① ORDINARY LEAST SQUARES (OLS): used to estimate the model parameters that minimizes the sum of the squared errors (SSE)
- OLS seeks to find the line that minimizes the sum of the squared distances between each training example & the line.
- Chooses the model parameters (slope & intercept of the line) in the way that the total area of all these squared distances is as small as possible.
- OLS can be viewed as minimizing the MSE (mean squared errors) all over the training examples in the training data.
- Noniterative method, training a linear regression model can be an iterative process.
  This iterative training process will use the gradient descent optimization algorithm used in logistic regression but will minimize a loss function used for regression S.A. MSE loss function.

DEMO: LINEAR REGRESSION

In this exercise we'll use scikit-learn to compute linear regression among some variables in the World Happiness Report (WHR) data set.

IMPORT PACKAGES

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
% matplotlib inline
pd.options.display.float_format = '{:.2f}'.format

from sklearn.linear_model import LinearRegression  # split function into training & test sets
from sklearn.model_selection import train_test_split  # split function into training & test sets
from sklearn.metrics import mean_squared_error, r2_score  # To evaluate our models
```

STEP 1: LOAD & PROCESS THE DATA

We'll just examine data from 2015 - 2019 & we'll store it in a dataframe called `df1517`.

```
dfraw = pd.read_excel('WHR2018Chapter2OnlineData.xls', sheet_name='Table2.1')
```

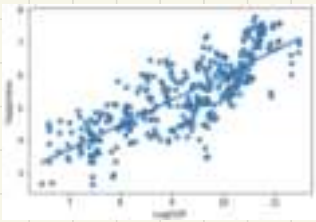
```
cols_to_include = ['country', 'year', 'Life Ladder',
                   'Positive affect','Negative affect',
                   'Log GDP per capita', 'Social support',
                   'Healthy life expectancy at birth',
                   'Freedom to make life choices',
                   'Generosity', 'Perceptions of corruption']
renaming = {'Life Ladder': 'Happiness',
            'Log GDP per capita': 'LogGDP',
            'Social support': 'Support',
            'Healthy life expectancy at birth': 'Life',
            'Freedom to make life choices': 'Freedom',
            'Perceptions of corruption': 'Corruption',
            'Positive affect': 'Positive',
            'Negative affect': 'Negative'}
df = dfraw[cols_to_include].rename(renaming, axis=1)
df1517 = df[df.year.isin(range(2015,2018))]
df1517 = df1517.dropna() # remove missing values
df1517.head()
```

The WHR is interested in self-reported Happiness is dependant on diff. factors they measure (S.A. LogGDP, Support, Life)

STEP 2: VISUALIZE VARIABLES OF INTEREST

Happiness & LogGDP are well correlated. We use scatterplot to analyze this relationship. And overlay a line of best fit.

```
sns.regplot(x="LogGDP", y="Happiness", data=df1517)
```



STEP 3: CREATE LABELED EXAMPLES FROM THE DATA SET

LogGDP ↓ Happiness ↓ } Our model will Predict a Happiness value for a given LogGDP value

Create a SIMPLE LINEAR REGRESSION model that finds the linear relationship between 1 FEATURE & 1 LABEL

Extract the LogGDP column from df1517 assign to variable x. This will be our FEATURE.
↳ Use scikit-learn to_frame() on x to keep it as Pandas DF instead of a series. This is IMPORTANT for the Linear Regression fit() method.

Extract the Happiness column from df1517. Assign to y. This will be our LABEL

Print values x & y

```
x= df1517['LogGDP'].to_frame()
y= df1517['Happiness']

print(X)
print(y)
```

STEP 4: CREATE TRAINING & TEST DATA SET

Now we have specified examples, split into training & test data set to estimate the model's parameter w_1 and a & test set to better understand the performance of our model on new data.

1. Use scikit-learn train_test_split() to create the data sets.
2. Specify:
   ↳ A test set is 15% of the size of the data set
   ↳ Seed value of 42

```
X_test, X_train, y_test, y_train= train_test_split(X,y, test_size= 0.15, random_state = 42)
```

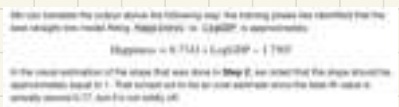
STEP 5: FIT A LINEAR REGRESSION MODEL WITH TRAINING SET

```
# Create the LinearRegression model object
# store into variable 'model'
model = LinearRegression()

# fit the model to the training data
model.fit(X_train, y_train)

# make predictions on the test data
# store into variable 'prediction'. We'll compare these values to y_test later
prediction = model.predict(X_test)
```

Inspect the model parameters that were determined during training.

The weights w are stored in numpy array `model.coef_`. To access particular weight s.a. w_1 we do: `model.coef_[0]`
The intercept a is stored in `model.intercept_`

```
# weight_1 (weight of feature LogGDP)
print("Model Summary)\n\n Weight_1 = ", model.coef_[0], "[weight of feature LogGDP]")

# alpha
print("Alpha = ", model.intercept_, "[intercept]")
```



STEP 6: EVALUATE THE MODEL ON TEST SET

Now examine how well the model performed.

To evaluate our model we'll compute the **RMSE (root mean square error)** to evaluate Regression models.
↳ Finds differences between the predicted values & actual values

To compute RMSE we use scikit-learn `mean_squared_error()` which computes the MSE between `y_test` and `prediction`. We then take the $\sqrt{}$ of result to get RMSE.

Use the **coefficient of determination AKA. R^2**
↳ Measures the porportion of variability in the prediction that the model was able to make using the input data.
↳ $R^2 = 1 \Longrightarrow$ perfect
$R^2 = 0 \Longrightarrow$ no explanatory value
↳ use scikit-learn `r2_score()` to compute it.

```
# The MSE
print("\n Model Performance\n\nRSME = %.2f"
      % np.sqrt(mean_squared_error(y_test, prediction)))  # 0.73

# The coefficient of determination: 1 is perfect prediction
print('R^2 = %.2f'
      % r2_score(y_test, prediction))  # 0.56
```

STEP 7: VISUALIZE THE MODEL

Plot should look similar to plot in STEP 2.

```
#set s to reasonable value to prevent overplotting of points on top of each other
plt.scatter(X, y, color="black", s=15);

# use plt.plot() to add a blue line to plot using the values in X_test & prediction
plt.plot(X_test, prediction, color='blue', linewidth=3);

plt.xlabel('LogGDP');
plt.ylabel('Happiness');
```

STEP 8: CREATE LABLED EXAMPLES FOR MULTIPLE LINEAR REGRESSION

- Assign variable features the list of column names for the features of interest.
- Assign y the happiness columns in df1517
- Assign x the columns in df1517 that are listed in features
  - ↳ Extracting multiple columns from df1517 the result will be a DF ∴ we don't need to .to_frame()

```
features = ['Log GDP', 'Support', 'Life', 'Freedom', 'Generosity', 'Corruption']
X= df1517[features]
y= df1517['Happiness']
```

STEP 9: CREATE TRAINING & TEST DATA SET

1. Use scikit-learn train_test_split() to create the data sets.
2. Specify:
   - ↳ A test set is 15% of the size of the data set
   - ↳ Seed value of 42

```
X_test, X_train, y_test, y_train= train_test_split(X,y, test_size = 0.15, random_state = 42)
```

STEP 10: FIT A MULTIPLE LINEAR REGRESSION MODEL

Create OLS model for multiple linear regression:

```
# Create the LinearRegression model object
# store into variable model2
model2 = LinearRegression()

# fit the model to the training data
model2.fit(X_train, y_train)

# make predictions on the test data
# Store into variable prediction2. We'll compare these values to y_test later
prediction2 = model2.predict(X_test)
```

Examine the parameters:

```
print("Model Summary:\n")

#print intercept (alpha)
print("Intercept: ")
print("alpha = ", model2.intercept_)

#print weights
print("\n Weights: ")
i=0
for w in model2.coef_:
    print('w_',i+1, "= ", w, '[weight of', features[i], ']')
    i+= 1
```

```
Model Summary:

Intercept:
alpha =  -2.424946874311782

Weights:
w_ 1 =  0.2950899282096664  [ weight of  LogGDP ]
w_ 2 =  2.846499816214751  [ weight of  Support ]
w_ 3 =  0.8121548440048818  [ weight of  Life ]
w_ 4 =  1.5133471841693855  [ weight of  Freedom ]
w_ 5 =  0.2902663701254363  [ weight of  Generosity ]
w_ 6 =  -0.5175437220884864  [ weight of  Corruption ]
```

STEP 11: EVALUATE THE MODEL ON THE TEST SET

```
# The MSE
print("\n Model Performance\n\nRSME = %.2f"
      % np.sqrt(mean_squared_error(y_test, prediction2)))
# The coefficient of determination: 1 is perfect prediction
print('R^2 = %.2f'
      % r2_score(y_test, prediction2))
```

```
Model Performance

RMSE =  0.62
 R^2 =  0.68
```

Step 12: Conclusions

Examine the output in the Model Summary. Estimates of model parameters are now provided for all of the features, as well as the overall intercept. Note that the estimates for the intercept and the 'LogGDP' weight are different than was the case in the simple regression. That is typical, since multiple regression accounts for relationships between each independent and dependent variable once all the other data relationships are taken into account.

We can see from the summary results that the 'Support' and 'Freedom' variables have the largest weights, indicating that, on average, an increase in those variables corresponds to an increase in 'Happiness'. 'Corruption' has a negative weight implying that, on average, a decrease in that feature corresponds to an increase in 'Happiness'. These results fit with our common sense which is always important to verify.

We also see that our RMSE has decreased and our R^2 value has increased, both good indicators that adding more features has increased the accuracy and fit of the model (although it is important to note that adding variables will always increase R^2 and that the magnitude of the increase may differ depending on the variable)

EXERCISE COMPLETE