

Name: Skylar Furey

Link to shared Colab Notebook: https://colab.research.google.com/drive/1L_GAJ4TfeA90j1hf9BjcSZRCXwlew7vy?usp=sharing

General directions for coding assignments are as follows. *Please avoid any unnecessary loops.* Use [broadcasting](#), and write [vectorized](#), efficient code. *Unless otherwise instructed*, please only use low-level mathematical operations (for example, math and linear algebra operators in numpy and tensorflow), and avoid using high-level implementations (for example, high-level implementations of methods from scikit-learn and keras). If you are unsure about the usage of any library/function, please contact the instructor.

✓ Not So Linear Regression [50 Total Points]

Often times linear regression is presented in terms of input features, which results in linear models (shocker!). Linear models are an already powerful technique, but they are super-charged when constructed atop of *nonlinear* features. Arguably, the construction of linear models with nonlinear features is one of the strongest (and most used) tricks in ML and is core to neural networks. Below we explore how the expressiveness of linear models increases based on features. This highlights the importance of the question, "*Linear in what?*"

✓ Data and Setup

First, it may help to enable GPUs for the notebook:

- Navigate to Edit→Notebook Settings
- select GPU from the Hardware Accelerator drop-down

Next, confirm that we can connect to the GPU with tensorflow.

(Note, it is fine if you cannot connect to GPU, it just might take a little longer to run.)

```
%tensorflow_version 2.x
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    print('GPU device not found')
else:
    print('Found GPU at: {}'.format(device_name))
```

↔ Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.
Found GPU at: /device:GPU:0

Let's import additional packages of use.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats
import pandas as pd
```

Generate data.

Generate N data samples of input (1d) features X and outputs Y , which shall depend non-linearly on X .

```
d = 1
N = 2000

X = tf.random.normal((N, 1))

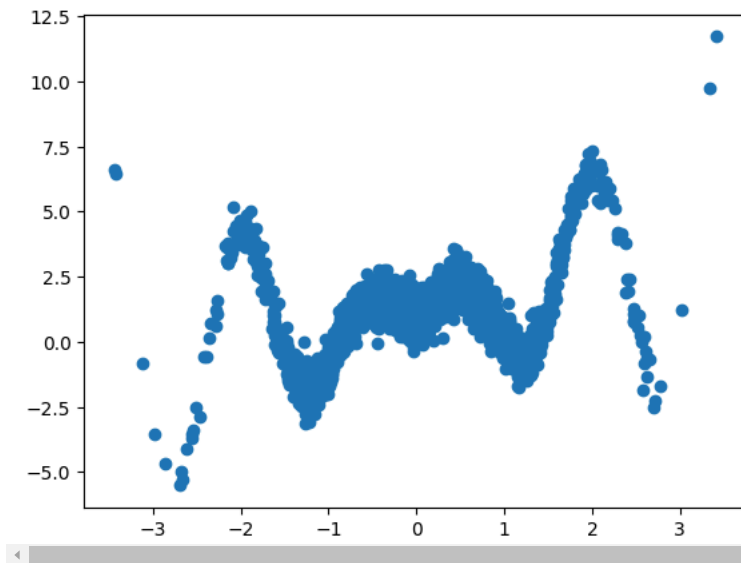
def f(x):
    return tf.exp(0.5*x)+2.0*x*tf.sin(4.0*x)

sigma = 0.5
Y = f(X) + sigma*tf.random.normal((N,1))
```

Visualize the data.

```
plt.scatter(X, Y)
```

```
<matplotlib.collections.PathCollection at 0x7f329dbacd60>
```



Linear Regression (20 Points)

[14 points] First let's implement a function that returns the linear coefficients for ordinary least squares regression.

```
def linear_coefs(X, Y):
    """
    Args:
        X: N x d matrix of input features
        Y: N x 1 matrix (column vector) of output response

    Returns:
        Beta: d x 1 matrix of linear coefficients
    """

    XtX = tf.matmul(X, X, transpose_a=True)
    XtY = tf.matmul(X, Y, transpose_a=True)

    beta = tf.linalg.solve(XtX, XtY)

    return beta
```

[6 points] Clearly, the groundtruth relationship is not a linear one. However, let's see what the best linear fit is for this data.

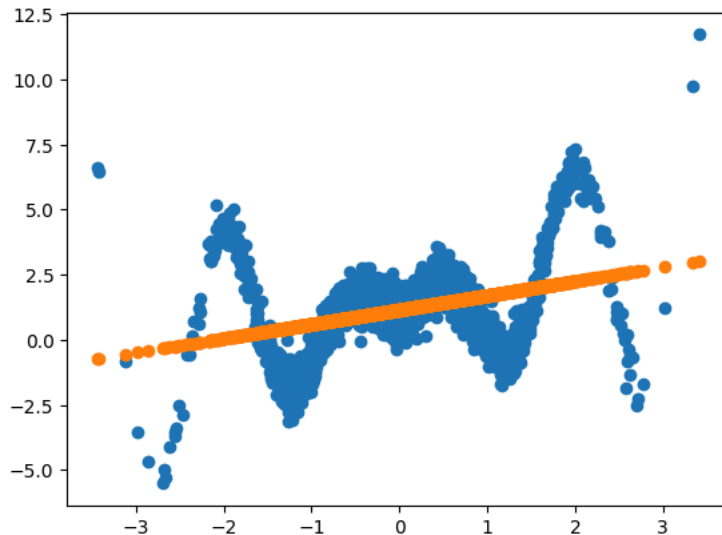
```
# TODO: be sure to account for an intercept term
interceptTerm = tf.ones((X.shape[0], 1))
XWithIntercept = tf.concat([interceptTerm, X], axis=1)

betaLinear = linear_coefs(XWithIntercept, Y)
Y_linear = tf.matmul(XWithIntercept, betaLinear)
```

Let's plot the linear predictions.

```
plt.scatter(X, Y)
plt.scatter(X, Y_linear)
```

 <matplotlib.collections.PathCollection at 0x7f32819740d0>



As you can see this captures some trend, but leaves a lot to be desired.

✓ Polynomial Regression (18 Points)

Let's turn back time and revisit an algebra course classic: polynomials! Recall that polynomials are made up of monomials, which are products of variable (features) taken to powers. With 1d data, these are terms like x^2 , x^7 , etc. When we fit a polynomial, we are looking for β coefficients to $\hat{f}(x) = \sum_{j=1}^m \beta_j x^j + \beta_0$. Does this look familiar? (It should!)

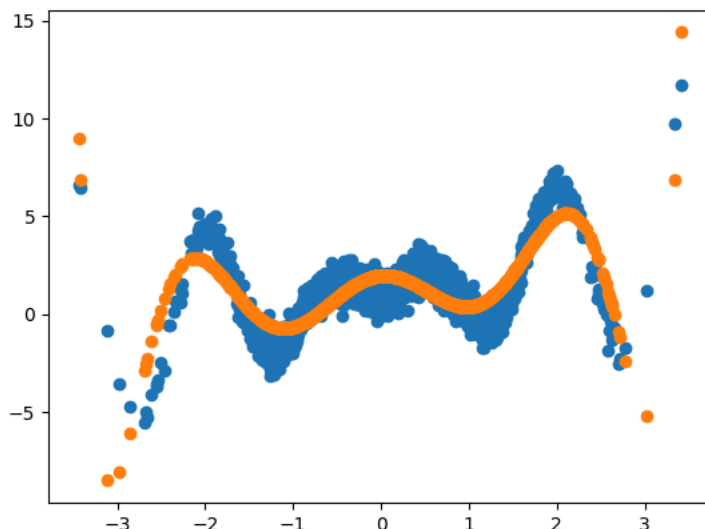
[18 points] Let's fit a polynomial using OLS. In particular, let's fit a polynomial of order 8.

```
# First generate the features corresponding to a 8th order 1d polynomial.
# You'll lose points :( if you do not do it in a simple, single line
XPoly = np.hstack([X**i for i in range(9)])
# Lets get the corresponding polynomial coefficients
# USE linear_coefs!
betaPoly = linear_coefs(XPoly, Y)
# Finally lets plot the estimates with the polynomial
Y_poly = tf.matmul(XPoly, betaPoly)
```

Let's plot the polynomial estimates.

```
plt.scatter(X, Y)
plt.scatter(X, Y_poly)
```

<matplotlib.collections.PathCollection at 0x7f3281376c50>



A much better fit! In the future we'll discuss how to choose hyperparameters to our model (such as the order of polynomials). For now, just play around with the order to see how estimates vary.

✓ Random-Feature Regression (12 Points)

Lastly, we'll consider a seemingly crazy idea: performing linear regression over *random* features. That is, we shall fit a model

$\hat{f}(x) = \sum_{j=1}^m \beta_j \phi_j(x)$. Where $\phi_j(x)$ are randomly constructed features. In particular, we shall construct $\phi_j(x) = \cos(\omega_j^T x + b_j)$ where ω_j was drawn from a Gaussian, and b_j from a Uniform distribution. Note that (ω_j, b_j) s are drawn randomly once, and are then *held fixed* for the remainder of their usage. (Think about what goes wrong if they are redrawn randomly each time.) Although this seems adhoc, a linear model on such random features are actually approximating a very flexible class of functions (see <https://people.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf>).

```
def rand_feats(X, D=1000, gamma=1.0):
    """
    Args:
        X: N x d matrix of input features
        D: Integer, number of random features
        gamma: Float, scale of frequencies

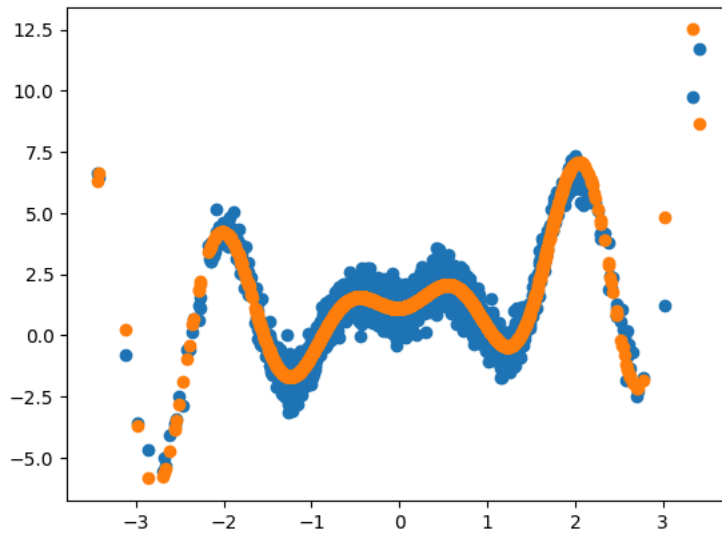
    Returns:
        Phis: N x D matrix of random features
    """
    d = X.get_shape()[1]
    tf.random.set_seed(112358) # Why is this needed?
    Ws = gamma*tf.random.normal((d, D))
    bs = 2.0*np.pi*tf.random.uniform((1, D))
    XWs = tf.matmul(X, Ws)
    return tf.cos(XWs+bs)
```

[7 points] Get predictions based on the random features above.

```
# TODO: Get the Y estimates using random features. (Default D and gamma is fine.)
randomFeatures = rand_feats(X)
betaRandFeats = linear_coefs(randomFeatures, Y)
Y_randfeats = tf.matmul(randomFeatures, betaRandFeats)

plt.scatter(X, Y)
plt.scatter(X, Y_randfeats)
```

 <matplotlib.collections.PathCollection at 0x7f3281701570>



Not bad for a bunch of random features! This, along with kernels, are pretty powerful concepts. I highly recommend looking further into [random features](#).

[5 points] Why is it necessary to set the random seed `tf.random.set_seed(112358)` in the function `rand_feats` above. (Think about what would go wrong if during inference if we do not do this.)

TODO: The seed must be set to ensure that the random features are the same each time, if we were to redraw the features each time then the model would be trained on different data each time causing inconsistent predictions.

✓ Naive Bayes vs Logistic Regression [50 Points]

We now will explore classification techniques. In the following we shall explore the difference between a generative model, Naive Bayes, and a discriminative model, Logistic Regression. Moreover, we will gain some experience solving for models *iteratively*, with gradient descent. Again, *avoid loops throughout* (i.e. write efficient code).


Reading data

Let's use some helper functions to read in pregenerated 2d data for classification. `X` and `Y` will be $N \times 2$ input features and $N \times 1$ binary output labels, respectively.

```
def read_dataset(url):
    data = pd.read_csv(url).to_numpy()
    X = data[:, :-1]
    Y = data[:, -1, None]
    return X.astype(np.float32), Y.astype(np.float32)

url1 = 'https://raw.githubusercontent.com/lupalab/comp755_f21/main/hw3_1.csv'
X, Y = read_dataset(url1)
```

`X.shape`

 (199, 2)

✓ Naive Bayes [20 points]

First we shall estimate a *generative* model using Naive Bayes and Gaussians.

[6 points] Estimating Naive Bayes Parameters

We will use a Naive Bayes assumption with Gaussian distributions (with unit variance and unknown means). That is we will assume that $p(x_i | y) = \mathcal{N}(x_i; \mu_y, 1)$, for features x_i . Estimate the parameters of this Naive Bayes Model below.

```
def estimate_naive_bayes(X, Y):
    """
    Estimates the paramters of the Gaussian Naive Bayes model.
    Args:
        X: Nxd matrix of input features
        Y: Nx1 vector of output labels
    Returns:
        pi1: real, class prior probability for class 1
        mu1: 1 x d vector, parameters for class 1
        mu0: 1 x d vector, parameters for class 0
    """

    pi1 = np.sum(Y == 1) / Y.shape[0]
    mu1 = np.mean(X[Y.flatten() == 1], axis = 0)
    mu0 = np.mean(X[Y.flatten() == 0], axis = 0)

    return pi1, mu1, mu0

pi1, mu1, mu0 = estimate_naive_bayes(X, Y)
```

[6 points] Generating with the Generative Model

We saw in class that Naive Bayes is a generative approach. We will use this model to now generatie synthetic data.

```
def generate_naive_bayes(N, pi1, mu1, mu0):
    """
    Generate synthetic samples according to Naive Bayes model
    Args:
        N: int, number of samples to generate
        pi1: real, class prior probability for class 1
        mu1: 1 x d vector, parameters for class 1
        mu0: 1 x d vector, parameters for class 0
    Returns:
        X: Nxd matrix, synthetic input feature samples
        Y: Nx1 vector, synthetic output label samples
    """

    Y = np.random.binomial(1, pi1, size=(N, 1))

    d = len(mu1) # also could use len(mu0)
    X = np.zeros((N, d))

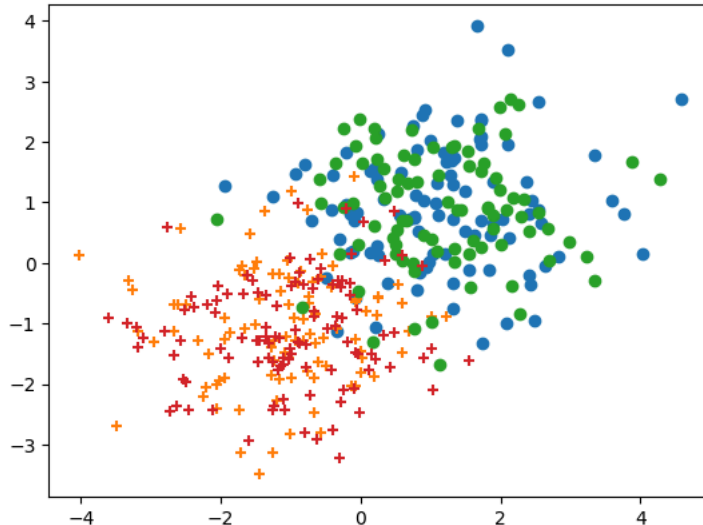
    X[Y.flatten() == 1] = np.random.normal(mu1, 1, size=(Y[Y.flatten() == 1].shape[0], d))

    X[Y.flatten() == 0] = np.random.normal(mu0, 1, size=(Y[Y.flatten() == 0].shape[0], d))

    return X, Y

Xhat, Yhat = generate_naive_bayes(200, pi1, mu1, mu0)
plt.scatter(X[np.equal(Y[:,-1], 0), 0], X[np.equal(Y[:,-1], 0), 1])
plt.scatter(X[np.equal(Y[:,-1], 1), 0], X[np.equal(Y[:,-1], 1), 1], marker='+')
plt.scatter(Xhat[np.equal(Yhat[:,-1], 0), 0], Xhat[np.equal(Yhat[:,-1], 0), 1])
plt.scatter(Xhat[np.equal(Yhat[:,-1], 1), 0], Xhat[np.equal(Yhat[:,-1], 1), 1], marker='+')
```

 <matplotlib.collections.PathCollection at 0x7f32817ec280>



Should look pretty good! (I.e. the generated data should match up with the training data.)

[6 points] Predicting with the Generative Model

In addition to generating synthetic data, we can also use the generative model to make predictions. We implement this below.

```
def naive_bayes_pred(x, pi1, mu1, mu0):
    """
    Make predictions using Naive Bayes model.
    Args:
        x: mxd matrix of input feature vectors for m queries
        pi1: real, class prior probability for class 1
        mu1: 1 x d vector, parameters for class 1
        mu0: 1 x d vector, parameters for class 0
    Returns:
        Yhat: mx1 vector of floats in {0.0, 1.0}, predicted output label samples
    """
    logLikelihoodClass1 = -0.5 * np.sum(((x - mu1) ** 2), axis=1)
    logLikelihoodClass0 = -0.5 * np.sum(((x - mu0) ** 2), axis=1)

    logOdds = logLikelihoodClass1 + np.log(pi1) - (logLikelihoodClass0 + np.log(1 - pi1))

    Yhat = (logOdds > 0).astype(float)

    return Yhat
```

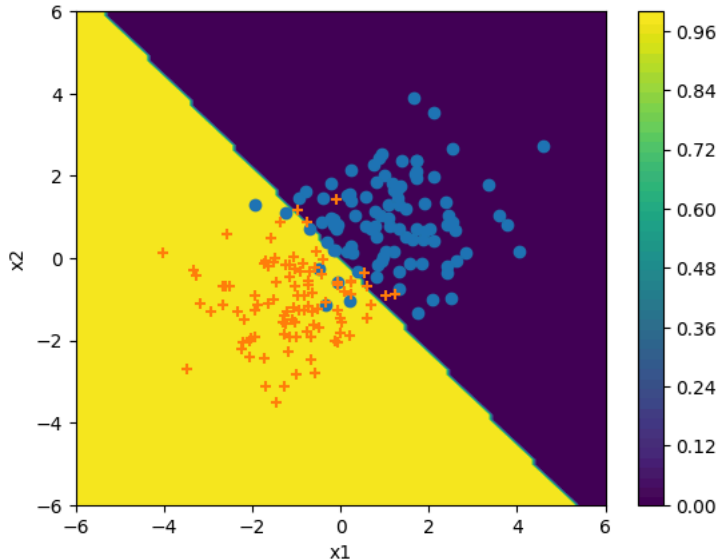
[1 points] Plot Decision Boundary

Next we'll use this helper function to plot our decision surface.

```
def plot_grid(pred_func):
    # Reduce the number of grid points if there is a memory issue
    gridx = np.float32(np.linspace(-6.0, 6.0, 100))
    gridx1, gridx2 = np.meshgrid(gridx, gridx)
    g1 = np.reshape(gridx1, [-1, 1])
    g2 = np.reshape(gridx2, [-1, 1])
    func_vals = pred_func(np.concatenate((g1, g2), -1))
    plt.contourf(gridx1, gridx2, np.reshape(func_vals, gridx1.shape), 64)
    plt.xlabel('x1')
    plt.ylabel('x2')
    cbar = plt.colorbar()
    cbar.solids.set_edgecolor('face')
    plt.draw()
    return func_vals, g1, g2

plot_grid(lambda x: naive_bayes_pred(x, pi1, mu1, mu0))
plt.scatter(X[np.equal(Y[:, -1], 0), 0], X[np.equal(Y[:, -1], 0), 1])
plt.scatter(X[np.equal(Y[:, -1], 1), 0], X[np.equal(Y[:, -1], 1), 1], marker='+')
```

 <matplotlib.collections.PathCollection at 0x7f329a262fb0>



[2 points] Computing Accuracy

Lastly, we shall compute the accuracy of predictions with a helper function.

```
def accuracy(X, Y, pred_func):
    """
    Helper function that returns the accuracy of a prediction function
    for given data.
    Args:
        X: Nxd matrix of input features
        Y: Nx1 vector of true output labels
        pred_func: function that takes in X and outputs Nx1 vector of predictions
    Returns:
        accu: float, accuracy of pred_func on (X, Y)
    """
    Y_pred = pred_func(X)

    Y_pred_binary = (Y_pred > 0.5).astype(float)

    accu = np.mean(Y_pred_binary.flatten() == Y.flatten())

    return accu
```

```
accu = accuracy(X, Y, lambda x: naive_bayes_pred(x, p11, mu1, mu0))
print('Naive Bayes Accuracy on Training Data: {}'.format(accu))
```

 Naive Bayes Accuracy on Training Data: 0.9296482412060302

✓ Logistic Regression [18 points]

We now move on to training a *discriminative* model, logistic regression. We will train a linear logistic regression model iteratively, using gradient descent.

[2 points] Predictions with Logistic Regression

First we shall implement how to make predictions using input features x and parameters β .

```
def logistic_pred(x, beta):
    """
    Return predictions of logistic regression model.
    Args:
        x: mxd matrix of input features
        beta: (d+1)x1 vector of model parameters where beta[-1, 0] is intercept term
    Returns:
        yhat: mx1 vector of floats in {0.0, 1.0}, predicted output label samples
    """
```



```
XWithIntercept = np.hstack((np.ones((x.shape[0], 1)), x))

linearCombination = np.dot(XWithIntercept, beta)

probs = 1 / (1 + np.exp(-linearCombination))

yhat = (probs >= 0.5)

return yhat
```

[8 points] Log Likelihood

Next we shall compute the mean log-likelihood according to the logistic model: $\frac{1}{N} \sum_{i=1}^N \log p_{\beta}(Y_i | X_i)$.

```
def mean_logistic_loglikelihood(X, Y, beta):
    """
    Returns mean loglikelihood.
    Args:
        X: Nxd matrix of input features
        Y: Nx1 vector of true output labels
        beta: (d+1)x1 vector of model parameters where beta[-1, 0] is intercept term
    Returns:
        mll: scalar of mean loglikelihood according to beta for data points
    """

    linearCombination = tf.matmul(X, beta)
    mll = tf.reduce_mean(Y * (-tf.nn.softplus(-linearCombination)) +
                        (1 - Y) * (-tf.nn.softplus(linearCombination)))

    return mll
```

[2 points] Compare Log Likelihood

Check your implementation against an implementation using tensorflow's library

```
XWithIntercept = tf.concat([tf.ones((tf.shape(X)[0], 1)), X], axis=1)

beta_init = (np.random.randn(3, 1)*0.1).astype(np.float32)
mll_implementation = mean_logistic_loglikelihood(XWithIntercept, Y, beta_init)

probs = tf.sigmoid(tf.matmul(XWithIntercept, beta_init))
mll_TF = -tf.reduce_mean(tf.keras.losses.binary_crossentropy(Y, probs))

print('Own implementation:\n{}\nTensorflow Implementation:\n{}'.format(
    mll_implementation, mll_TF
))
```

```
↔ Own implementation:
-0.6530263423919678
Tensorflow Implementation:
-0.6530263423919678
```

[3 points] Compute Logistic Regression Gradient

Next, we shall implement the gradient to the `mean_logistic_loglikelihood` as a function of `beta`. We'll compare to using Tensorflow to compute this gradient automatically; but first, let's roll up our sleeves! (*Note: the gradient is derived in the slides, no need to rederive!*)

```
def mean_logistic_loglikelihood_gradient(X, Y, beta):
    """
    Returns the gradient of mean log-likelihood loss
    (i.e. negative mean log-likelihood) w.r.t. beta.
    Args:
        X: Nxd matrix of input features
        Y: Nx1 vector of true output labels
        beta: (d+1)x1 vector of current model parameters
            where beta[-1, 0] is intercept term
    Returns:
        beta_grad: (d+1)x1 vector of gradients (partial derivatives) evaluated at beta
    """

    # XWithIntercept = tf.concat([tf.ones((tf.shape(X)[0], 1)), X], axis=1)

    # print(X.shape)
```

```

# print(X.shape)
# print(beta.shape)

linear_combination = tf.matmul(X, beta)

probabilities = tf.sigmoid(linear_combination)

error = Y - probabilities

# print(tf.transpose(X).shape)
# print(error.shape)

beta_grad = -tf.matmul(tf.transpose(X), error) / tf.cast(tf.shape(X)[0], dtype=X.dtype)

return beta_grad

```

Check your implementation, it should be *very* close to the same answer that Tensorflow's autodiff provides.

```

def mean_logistic_loglikelihood_tfgrad(X, Y, beta):
    with tf.GradientTape() as tape:
        loss_value = -mean_logistic_loglikelihood(X, Y, beta)
        g = tape.gradient(loss_value, [beta])[0]
    return tf.cast(g, tf.float32)

beta_init = (np.random.randn(3, 1)*0.1).astype(np.float32)
beta_grad = mean_logistic_loglikelihood_gradient(X, Y, beta_init)

beta_var = tf.Variable(shape=(3,1), initial_value=beta_init)
beta_tfgrad = mean_logistic_loglikelihood_tfgrad(X, Y, beta_var)

print('Own implementation:\n{}\nautograd:\n{}\nnorm: {}'.format(
    beta_grad.numpy(), beta_tfgrad.numpy(),
    tf.norm(beta_grad-beta_tfgrad, axis=0)))

```



InvalidArgumentError Traceback (most recent call last)
 <ipython-input-29-4539ebf8e4dd> in <cell line: 8>()

```

6
7 beta_init = (np.random.randn(3, 1)*0.1).astype(np.float32)
----> 8 beta_grad = mean_logistic_loglikelihood_gradient(X, Y, beta_init)
9
10 beta_var = tf.Variable(shape=(3,1), initial_value=beta_init)

```

3 frames

```

/usr/local/lib/python3.10/dist-packages/tensorflow/python/framework/ops.py in raise_from_not_ok_status(e, name)
5981 def raise_from_not_ok_status(e, name) -> NoReturn:
5982     e.message += (" name: " + str(name if name is not None else ""))
-> 5983     raise core._status_to_exception(e) from None # pylint: disable=protected-access
5984
5985

```

InvalidArgumentError: {{function_node __wrapped__ MatMul_device_/job:localhost/replica:0/task:0/device:GPU:0}} Matrix size-incompatible: In[0]: [199.21]. In[1]: [3.1] [Op:MatMul] name:

Next steps: [Explain error](#)

We'll train the logistic regression model with gradient descent. You will have the `gradient_descent` function implemented, but you'll have to figure out the right argument/parameters to feed into it to train an effective model.

```

def gradient_descent(beta_init, lossfunc, gradfunc,
                    t=5e-5, lr_decay=1e-2, nsteps=100):
    """
    Run gradient descent for n steps starting from beta_init with t step size,
    which is decayed lr_decay much each iteration.
    Args:
        beta_init: (d+1)x1 vector of initial model parameters to begin
            gradient descent from
        lossfunc: (d+1)x1 -> scalar function that we are minimizing
        gradfunc: (d+1)x1 -> (d+1)x1 function that computes gradients at current
            parameter
        t: float, step size for updates
        lr_decay: float, amount to decay t by after each step `t *= (1-lr_decay)`
        nsteps: int, number of updates to make
    Returns:
    """

```

```

    beta_var: (d+1)x1 vector variable with optimized values
    """
    beta_var = tf.Variable(shape=beta_init.shape, initial_value=beta_init)

    for i in range(0, nsteps):
        beta_var.assign(beta_var - t*gradfunc(beta_var))
        t *= (1-lr_decay)
        plt.scatter(i, lossfunc(beta_var), marker='.')

    return beta_var

```

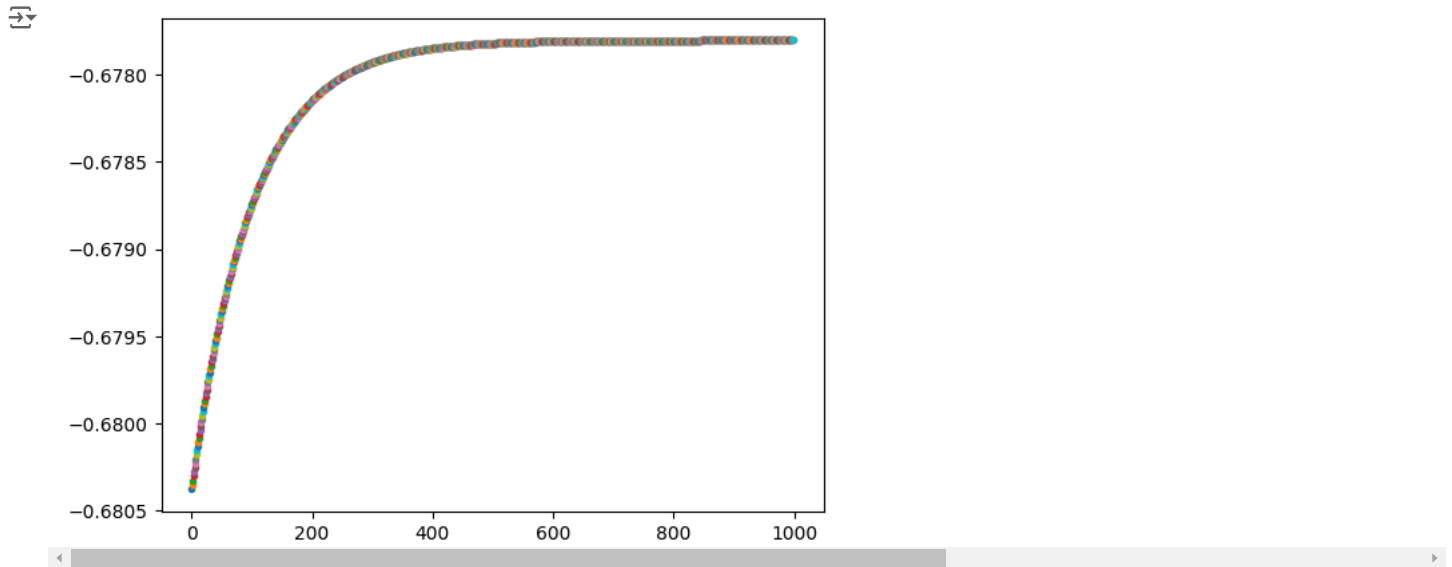
[3 points] Train your model

Use the `gradient_descent` function to optimize a good logistic regression classifier. Note: *you will have to play with some of the parameters of gradient descent.*

```

lossfunc = lambda beta: mean_logistic_loglikelihood(XWithIntercept, Y, beta)
gradfunc = lambda beta: mean_logistic_loglikelihood_gradient(XWithIntercept, Y, beta)
beta_opt = gradient_descent(beta_init, lossfunc, gradfunc, nsteps=1000)

```



[1 points] Plot Decision Boundary

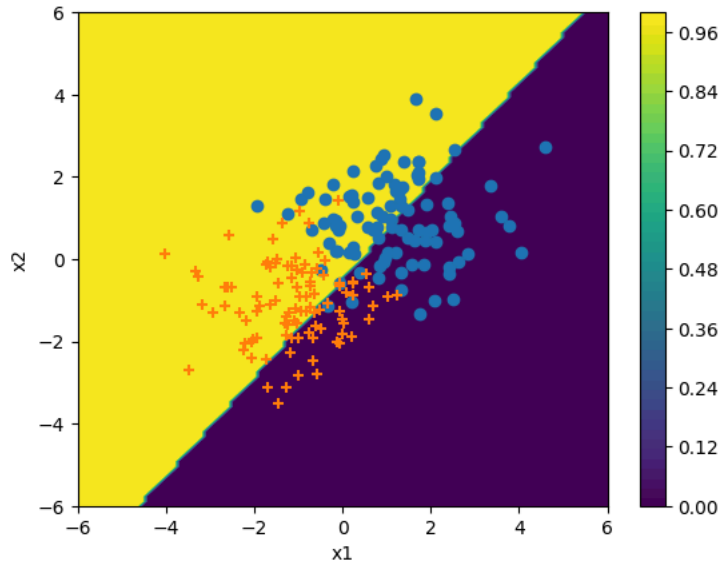
Plot the decision surface for the learned logistic regression model

```

plot_grid(lambda x: logistic_pred(x, beta_opt))
plt.scatter(X[np.equal(Y[:,-1], 0), 0], X[np.equal(Y[:,-1], 0), 1])
plt.scatter(X[np.equal(Y[:,-1], 1), 0], X[np.equal(Y[:,-1], 1), 1], marker='+')

```


 <matplotlib.collections.PathCollection at 0x7f328188df90>



[1 points] Compute accuracy

Compute the accuracy of the learned logistic regression model. (Hint: you should see over 94% accuracy on training data.)

```
accu = accuracy(X, Y, lambda x: logistic_pred(x, beta_opt))
print('Logistic Regression Accuracy on Training Data: {}'.format(accu))
```

 Logistic Regression Accuracy on Training Data: 0.542713567839196

▼ A Tricky Case

Above we saw that both Naive Bayes and logistic regression performed well. Next, we will look at a case where this is no longer true and we will assess what went wrong.

```
url2 = 'https://raw.githubusercontent.com/lupalab/comp755_f21/main/hw3_2.csv'
X, Y = read_dataset(url2)
```

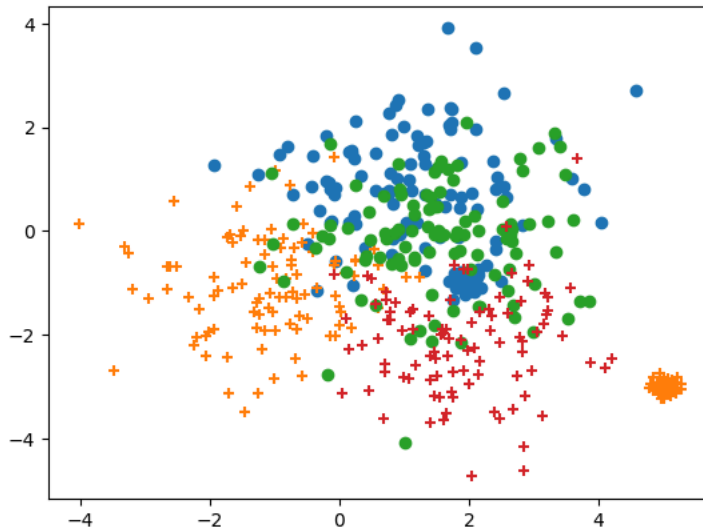
▼ Naive Bayes

First, let's see how Naive Bayes performs on the new dataset. We'll estimate parameters, and draw samples.

```
pi1, mu1, mu0 = estimate_naive_bayes(X, Y)
```

```
Xhat, Yhat = generate_naive_bayes(200, pi1, mu1, mu0)
plt.scatter(X[np.equal(Y[:,-1], 0)], X[np.equal(Y[:,-1], 0), 1])
plt.scatter(X[np.equal(Y[:,-1], 1)], X[np.equal(Y[:,-1], 1), 1], marker='+')
plt.scatter(Xhat[np.equal(Yhat[:,-1], 0)], Xhat[np.equal(Yhat[:,-1], 0), 1])
plt.scatter(Xhat[np.equal(Yhat[:,-1], 1)], Xhat[np.equal(Yhat[:,-1], 1), 1], marker='+')
```

 <matplotlib.collections.PathCollection at 0x7f320792edd0>



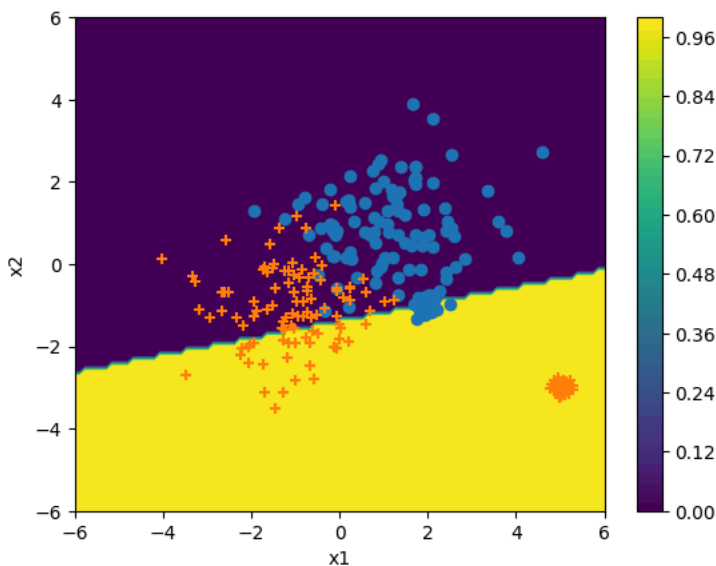
[2 points] Decision Surface and Accuracy

Let's now plot the decision surface and compute the training accuracy as before.

```
plot_grid(lambda x: naive_bayes_pred(x, pi1, mu1, mu0))
plt.scatter(X[np.equal(Y[:, -1], 0), 0], X[np.equal(Y[:, -1], 0), 1])
plt.scatter(X[np.equal(Y[:, -1], 1), 0], X[np.equal(Y[:, -1], 1), 1], marker='+')
```

```
accu = accuracy(X, Y, lambda x: naive_bayes_pred(x, pi1, mu1, mu0))
print('Naive Bayes Accuracy on Training Data: {}'.format(accu))
```

 Naive Bayes Accuracy on Training Data: 0.6741854636591479



Notice any differences from last time?

✓ Logistic Regression

[2 points] Train and Assess

We shall now optimize logistic regression for the new data, plot the decision surface, and get the training data accuracy (as before).

```
XWithIntercept = tf.concat([tf.ones((tf.shape(X)[0], 1)), X], axis=1)
lossfunc = lambda beta: mean_logistic_loglikelihood(XWithIntercept, Y, beta)
gradfunc = lambda beta: mean_logistic_loglikelihood_gradient(XWithIntercept, Y, beta)
beta_opt = gradient_descent(beta_init, lossfunc, gradfunc, nsteps=1000)
```

```
plt.figure()  
plot_grid(lambda x: logistic_pred(x, beta_opt))
```