Assignment 3 - Supervised Learning

Anna Berman

Netid: *aeb100*

1

[40 points] From theory to practice: classification through logistic regression

Introduction

For this problem you will derive, implement through gradient descent, and test the performance of a logistic regression classifier for a binary classification problem.

In this case, we'll assume our logistic regression problem will be applied to a two dimensional feature space. Our logistic regression model is:

$$f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}_i)$$

where the sigmoid function is defined as $\sigma(x) = \frac{e^x}{1+we^x}$. Also, since this is a two-dimensional problem, we define $\mathbf{w}^T \mathbf{x}_i = w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2}$ and here, $x_{i,0} \triangleq 1$

As in class, we will interpret the response of the logistic regression classifier to be the likelihood of the data given the model. For one sample, $(y_i, \mathbf{x_i})$, this is given as:

$$P(Y = y_i | X = x_i) = f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}_i)$$

Find the cost function that we can use to choose the model parameters, $\mathbf{w}$, that best fit the training data.

(a) What is the likelihood function of the data that we will wish to maximize?

For each x_i:

$P(Y = 1 | x_i w) = \sigma(\mathbf{w}^T \mathbf{x}_i)$
$P(Y = 0 | x_i w) = 1 - \sigma(\mathbf{w}^T \mathbf{x}_i)$

Therefore the cost fuction for the entirety of our data is:

$C(\mathbf{w}) = \Pi_{i=1}^{N} \sigma(\mathbf{w}^T \mathbf{x}_i)^{y_i} [1 - \sigma(\mathbf{w}^T \mathbf{x}_i)]^{1-y_i}$

$C(\mathbf{w}) = \Pi_{i=1}^{N} \hat{y}_i^{y_i} [1 - \hat{y}_i]^{1-y_i}$ for $\hat{y}_i \triangleq \sigma(\mathbf{w}^T \mathbf{x}_i)$

(b) Since a logarithm is a monotonic function, maximizing the $f(x)$ is equivalent to maximizing $\ln[f(x)]$. Express part (a) as a cost function of the model parameters, $C(\mathbf{w})$, that is the negative of the logarithm of (a).

$-log(C(\mathbf{w})) = -log(\Pi_{i=1}^{N} \hat{y}_i^{y_i} [1 - \hat{y}_i]^{1-y_i})$
$= -\Sigma_{i=1}^{N} log(\hat{y}_i^{y_i} [1 - \hat{y}_i]^{1-y_i})$
$= -\Sigma_{i=1}^{N} y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)$

The function above is the cost for each obervation in our dataset. We multiply by $\frac{1}{N}$ to finalize our cost function:
$= \frac{-1}{N} \Sigma_{i=1}^{N} y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)$ for $\hat{y}_i \triangleq \sigma(\mathbf{w}^T \mathbf{x}_i)$

(c) Calculate the gradient of the cost function with respect to the model parameters $\nabla_{\mathbf{w}} C(\mathbf{w})$. Express this in terms of the partial derivatives of the cost function with respect to each of the parameters, e.g. $\nabla_{\mathbf{w}} C(\mathbf{w}) = \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right]$.

$\frac{\partial C}{\partial w_0} = \frac{1}{N} \Sigma_{i=1}^{N} [\sigma(w_0 x_{i,0}) - y_i] x_{i,0}$
$\frac{\partial C}{\partial w_1} = \frac{1}{N} \Sigma_{i=1}^{N} [\sigma(w_1 x_{i,1}) - y_i] x_{i,1}$
$\frac{\partial C}{\partial w_2} = \frac{1}{N} \Sigma_{i=1}^{N} [\sigma(w_2 x_{i,2}) - y_i] x_{i,2}$

(d) Write out the gradient descent update equation, assuming η represents the learning rate.

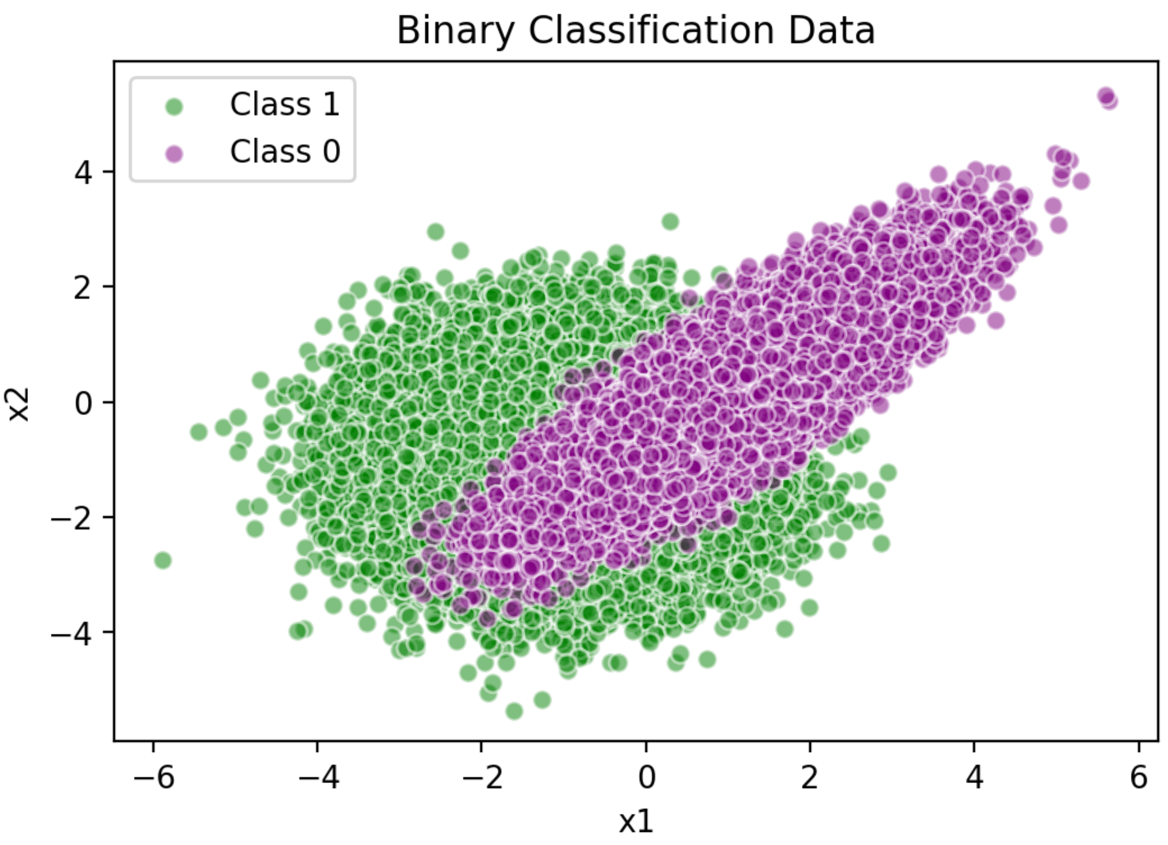
$w^{i+1} = w^i - \eta \frac{1}{N} \Sigma_{i=1}^{N} [\sigma(\mathbf{w}^T x_i) - y_i] x_i$

Prepare and plot your data

(e) Load the data and scatter plot the data by class. In the data folder in the same directory of this notebook, you'll find the data in `A3_Q1_data.csv`. This file contains the binary class labels, y, and the features x_1 and x_2. Comment on the data: do the data appear separable? Why might logistic regression be a good choice for these data or not?

At first glance, there does not appear to be a significant class imbalance. The data classes are split in what seems to be two overlapping ellipses. Although this pattern is easy to see with the human eye, from classification and prediction perspective these classes may be hard to model because there is a significant amount of overlap between the two classes. We move forward with logistic regression because of the binary nature of the classes. Logistic regression will also allow us to plot a decision boundary on our outcome space.

In [4]:
```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from math import e
import warnings
warnings.filterwarnings('ignore')

# Load data
df = pd.read_csv('./Assignment Code/Assignment 3/A3_Q1_data.csv')
x = df[["x1", "x2"]].values
y = df["y"].values

# Plot the data
plt.figure(dpi = 200)
plt.scatter(x[y == 1, 0], x[y == 1, 1], c='green', edgecolors='w',
            alpha = .5, label = 'Class 1')
plt.scatter(x[y == 0, 0], x[y == 0, 1], c='purple', edgecolors='w',
            alpha = .5, label = 'Class 0')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Binary Classification Data')
plt.legend()
plt.show()
```



(f) Do the data require any preprocessing due to missing values, scale differences, etc? If so, how did you remediate this?

There does not appear to be any missing values in the data and there does not appear to be any significant differences in scale between our two predictors. For those reasons we move forward without any preprocessing.

In [5]:
```
print(df.isnull().sum())
print(df.describe())
```

```
x1    0
x2    0
y     0
dtype: int64
                  x1             x2             y
count  100000.000000  100000.000000  100000.000000
mean        0.048529      -0.397106       0.500000
std         1.452409       1.164015       0.500003
min        -5.886436      -5.352265       0.000000
25%        -1.041007      -1.206343       0.000000
50%         0.039551      -0.401099       0.500000
75%         1.143536       0.402899       1.000000
max         5.634476       5.317718       1.000000
```

Implement gradient descent and your logistic regression algorithm

(g) Create a function or class to implement your logistic regression. It should take as inputs the model parameters, $\mathbf{w} = [w_0, w_1, w_2]$, and output the class confidence probabilities, $P(Y = y_i | X = x_i)$.

(h) Create a function that computes the cost function $C(\mathbf{w})$ for a given dataset and corresponding class labels.

(i) Create a function or class to run gradient descent on your training data. We'll refer to this as "batch" gradient descent since it takes into account the gradient based on all our data at each iteration (or "epoch") of the algorithm. Divide your data into a training and testing set where the test set accounts for 30 percent of the data and the training set the remaining 70 percent. In doing this we'll need to make some assumptions / experiment with the following:

1. The initialization of the algorithm - what should you initialize the model parameters to? For this, randomly initialize the weights to a different values between 0 and 1.
2. The learning rate - how slow/fast should the algorithm proceed in the direction opposite the gradient? This you will experiment with.
3. Stopping criteria - when should the algorithm be finished searching for the optimum? Set this to be when the cost function changes by no more than 10^{-6} between iterations. Since we have a weight vector, you can compute this by seeing if the L2 norm of the weight vector changes by no more than 10^{-6} between iterations.

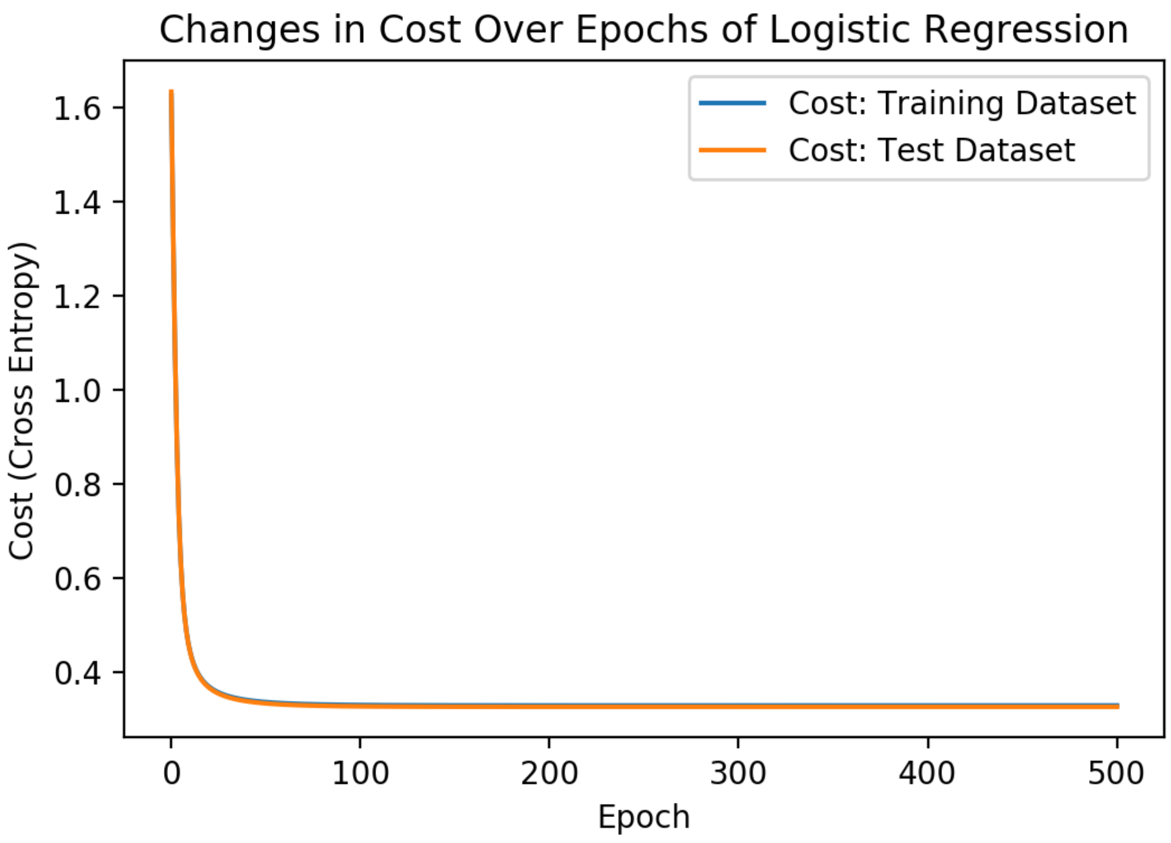
In [6]:
```
# Logistic Regression Classifier
#  Uses cross entropy as a cost function
#  Uses gradient decent as an optimizer
class LogisticReg:

    # Initialize classifer with weights w
    def __init__(self, w = None):
        self.w = w
        self.train_cost_history = []
        self.test_cost_history = []
        self.epoch = None

    # Sigmoid function
    # used in cost function and gradient decent
    def sigmoid(self, a):
        return 1 / (1 + e**(-a))

    # Cost function
    # Calculates cross entropy for predictors X and labels Y
    def cost(self, X, Y):
        N = len(Y)
        Y_hat = self.sigmoid(X.dot(self.w))
        cost = - Y.T.dot(np.log(Y_hat)) - (1-Y).T.dot(np.log(1- Y_hat))
        return cost / N

    # Calculate gradient for predictors X and labels Y
    def gradient(self, X, Y):
        N = len(Y)
        Y_hat = self.sigmoid(X.dot(self.w))
        return (1/N) * X.T.dot(Y_hat - Y)

    # Fit data
    # Takes in training data, test data, a learning, and a total number
    # of epochs
    def fit(self, x_train, y_train, x_test, y_test, learning_rate,
            epoch_total=500, printer = False):
        # Initialize weight to random observation
        self.w = np.random.rand(x_train.shape[1])

        self.epoch = 0
        while self.epoch <= epoch_total:
            current_w = self.w

            # Calculate cost functions for training set and append to
            # cost history
            cost_train = self.cost(x_train, y_train)
            self.train_cost_history.append(cost_train)

            # Calculate cost functions for test set and append to cost
            # history
            cost_test = self.cost(x_test, y_test)
            self.test_cost_history.append(cost_test)

            # Print progress
            if ((self.epoch + 1) % 100)  == 0 and (printer == True):
                print('Epoch: {0}'.format(self.epoch+1))
                print('  Train Data Cost: {0:0.4f}'.format(cost_train))
                print('  Test Data Cost: {0:0.4f}\n\n'.format(cost_test
))
                pass

            # Update weights
            y_hat = self.sigmoid(x_train.dot(self.w))
            gradient = self.gradient(x_train, y_train)
            self.w = current_w - learning_rate * gradient

            # Stop iterating if cost functino changes by less than 10^-6
            L2_norm_old = np.abs(np.linalg.norm(current_w, ord=2))
            L2_norm_new = np.abs(np.linalg.norm(self.w, ord=2))
            abs_val = np.abs(L2_norm_new - L2_norm_old)
            if (self.epoch != 0) and (abs_val < 1e-06) :
                break
            pass

            # Update the epoch number
            self.epoch += 1
        pass

    # Make a prediction given a set of preditors
    def predict(self, x):
        return self.sigmoid(x.dot(self.w))

    pass
```

In [7]:
```
# Separate x and y values from dataset
x = df[['x1', 'x2']].values
y = df['y'].values

# Divide data into a training and testing set
# Test set accounts for 30% of the data
# Training set the remaining 70%

# Shuffle order order of dataset
N = len(y)
shuffle = np.arange(N)
np.random.shuffle(shuffle)
x = x[shuffle]
y = y[shuffle]

# Partition on a 70:30 split
split = .7
break_point = int(split*N)
x_train = x[:break_point]
y_train = y[:break_point]
x_test = x[break_point:]
y_test = y[break_point:]

# Add intercept term
x_train = np.column_stack((np.ones(len(x_train)), x_train))
x_test = np.column_stack((np.ones(len(x_test)), x_test))
```

(j) At each step in the gradient descent algorithm it will produce updated parameter estimates. For each set of estimates, calculate the cost function for both the training and the test data.

In [8]:
```
# Fit a logistic regression
log_reg = LogisticReg()
log_reg.fit(x_train, y_train, x_test, y_test, learning_rate = .25,
            printer = True)

# Plot the change in cost function
plt.figure(dpi = 200)
x_range = range(log_reg.epoch)
plt.plot(x_range, log_reg.train_cost_history,
         label = 'Cost: Training Dataset')
plt.plot(x_range, log_reg.test_cost_history,
         label = 'Cost: Test Dataset')
plt.xlabel('Epoch')
plt.ylabel('Cost (Cross Entropy)')
plt.title('Changes in Cost Over Epochs of Logistic Regression')
plt.legend()
plt.show()
```

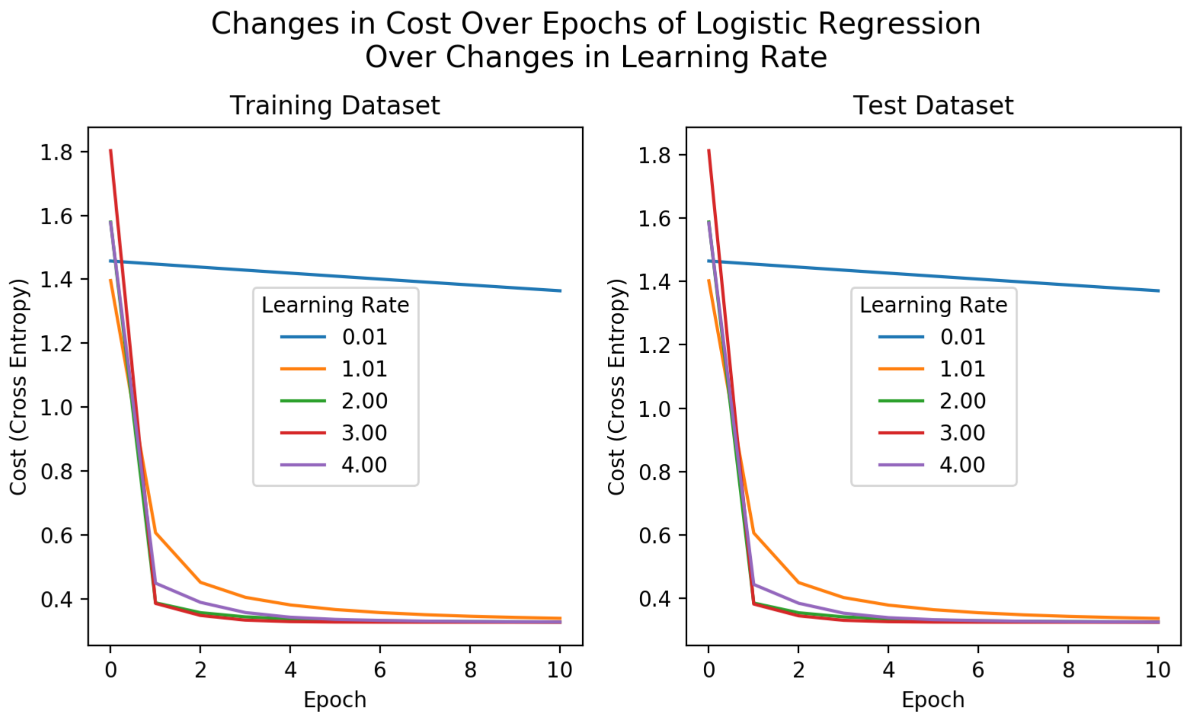
```
Epoch: 100
  Train Data Cost: 0.3293
  Test Data Cost: 0.3264


Epoch: 200
  Train Data Cost: 0.3283
  Test Data Cost: 0.3254


Epoch: 300
  Train Data Cost: 0.3283
  Test Data Cost: 0.3253


Epoch: 400
  Train Data Cost: 0.3283
  Test Data Cost: 0.3253


Epoch: 500
  Train Data Cost: 0.3283
  Test Data Cost: 0.3253
```



(k) Show this process for different learning rates by plotting the resulting cost as a function of iteration (or "epoch"). What is the impact that each parameter has on the process and the results? What choices did you make in your chosen approach and why? Use the parameter you choose here for the learning rate for the remainder of this question.

Changing the learning rate effected how quickly the cost function of the logistic regression dropped to it's minimum. From our plots it appears that a learning rate of 3 achieves the quickly cost decent.

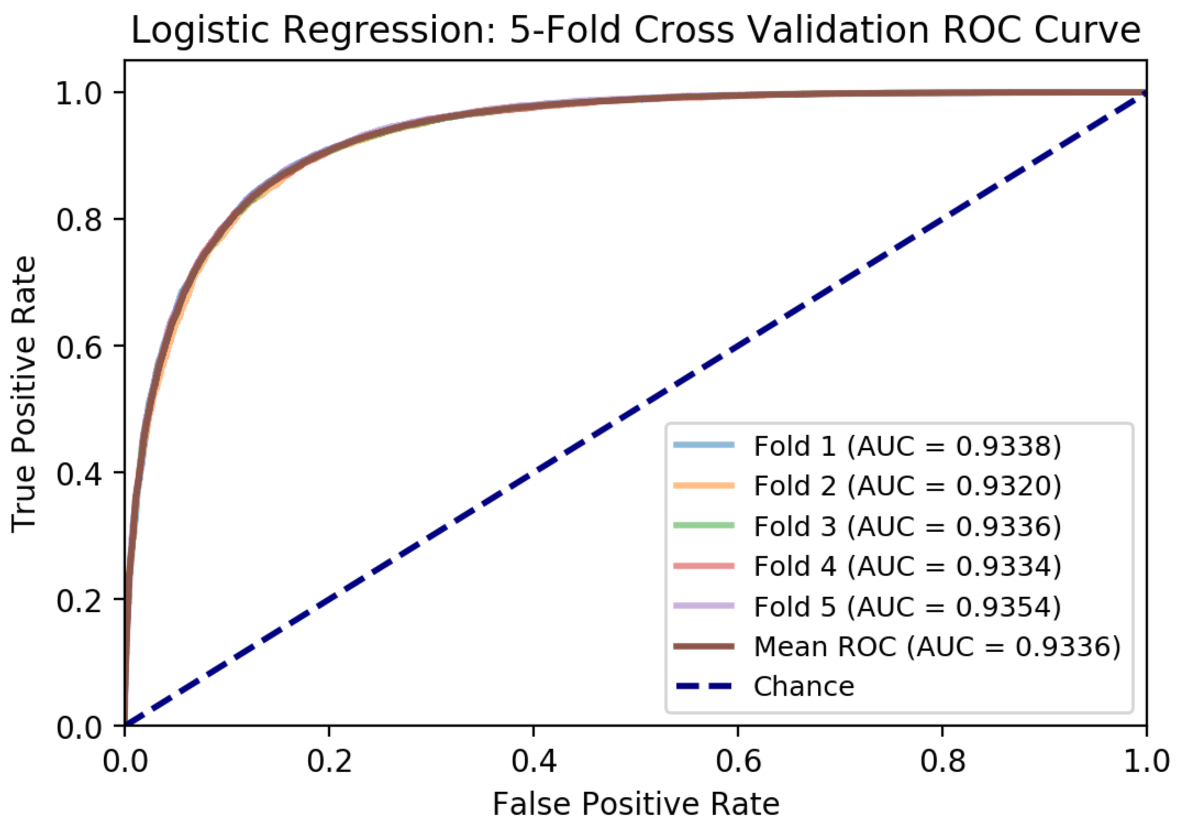
In [9]:
```
fig, axs = plt.subplots(1, 2)
fig.set_dpi(200)
fig.set_figheight(5)
fig.set_figwidth(8)
x_range = range(500)

# Iterate over various learning rates
# Plotting changes in costs over epochs
for learn_rate in np.linspace(.01,4,5):
    # Create instance of logistic regression
    log_reg = LogisticReg()

    # Fit logisitic regression, iterating over learning rate
    log_reg.fit(x_train, y_train, x_test, y_test,
                learning_rate = learn_rate, epoch_total = 10)

    # Plot the change in cost function
    x_range = range(log_reg.epoch)
    axs[0].plot(x_range, log_reg.train_cost_history,
                label = '{0:0.2f}'.format(learn_rate))
    axs[1].plot(x_range, log_reg.test_cost_history,
                label = '{0:0.2f}'.format(learn_rate))
    pass

# Format plot
axs[0].set_xlabel('Epoch')
axs[0].set_title('Training Dataset')
axs[0].set_ylabel('Cost (Cross Entropy)')
axs[0].legend(title = 'Learning Rate', loc ='best')
axs[1].set_xlabel('Epoch')
axs[1].set_title('Test Dataset')
axs[1].set_ylabel('Cost (Cross Entropy)')
axs[1].legend(title = 'Learning Rate')
fig.suptitle('Changes in Cost Over Epochs of Logistic Regression\n' +
             'Over Changes in Learning Rate', fontsize = 14)
plt.tight_layout(rect=[0, 0.03, 1, .90])
plt.show()
```



Test your model performance through cross validation

(l) Test the performance of your trained classifier using K-folds cross validation (while this can be done manually, the scikit-learn package StratifiedKFolds (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html#sklearn.model_selection.Stratif may be helpful). Produce Receiver Operating Characteristic Curves (ROC curves) of your cross validated performance.

In [10]:
```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score
plt.figure(dpi = 200)

# Cross validation
cross_val = StratifiedKFold(n_splits=5)
y_test_all = []
y_pred_all = []

# Loop through each fold
i = 0
for train, test in cross_val.split(x, y):
    # Create fold training and test sets
    x_train, x_test = x[train], x[test]
    y_train, y_test = y[train], y[test]
    # Append y test set to grand list
    y_test_all.append(y_test)

    # Add intercept term
    x_train = np.column_stack((np.ones(len(x_train)), x_train))
    x_test = np.column_stack((np.ones(len(x_test)), x_test))

    # Fit Logistic Regression
    log_reg = LogisticReg()
    log_reg.fit(x_train, y_train, x_test, y_test, learning_rate = 3)

    # Create preditions and append
    y_pred = log_reg.predict(x_test)
    y_pred_all.append(y_pred)

    # ROC Curve
    fpr, tpr, thresholds = roc_curve(y_test, y_pred)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw = 2, alpha = 0.5,
             label='Fold {0} (AUC = {1:0.4f})'.format(i+1, roc_auc))

    i += 1
    pass

# Mean AUC
y_test_all = np.concatenate(y_test_all)
y_pred_all = np.concatenate(y_pred_all)
fpr, tpr, thresholds = roc_curve(y_test_all, y_pred_all)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, lw = 2,
             label='Mean ROC (AUC = {0:0.4f})'.format(roc_auc))

# Format ROC Plot
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
         label = 'Chance')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right", fontsize = 9)
plt.title('Logistic Regression: {}-Fold Cross Validation ROC Curve'.form
at(i))
plt.show()
```



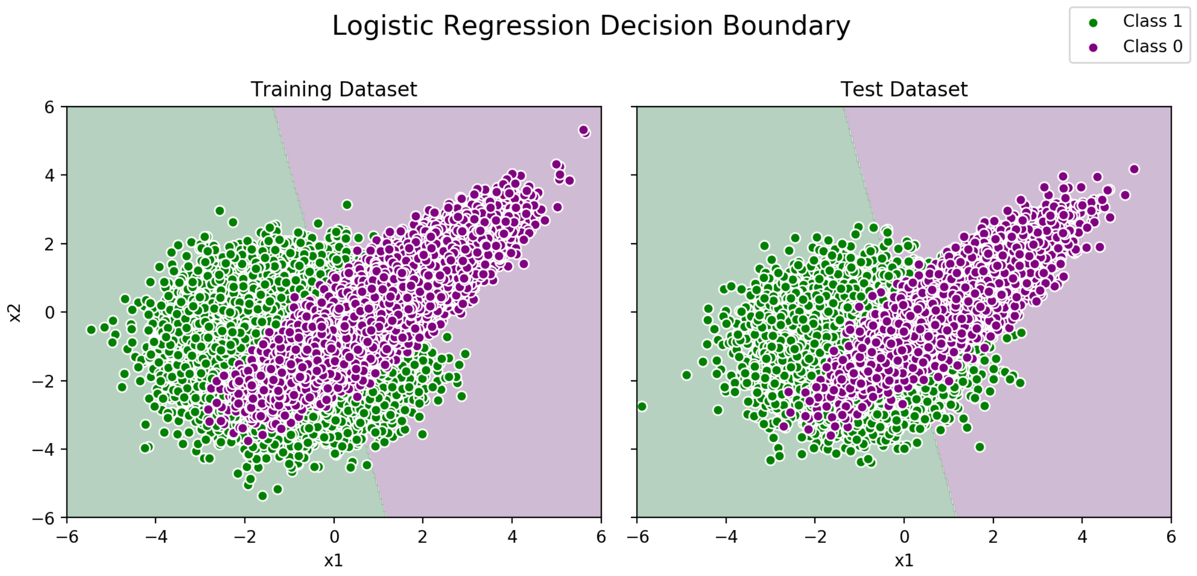
(m) Why do we use cross validation?

We use cross validation as a way to increase the accuracy of our estimates of out-of-sample error. By creating multiple 'training' and 'test' (or validation) sets within our original dataset, we can estimate error of our model on each fold. The average across each fold to is a more reliable estimate of out-of-sample error because it's not reliant on the way a single, training/test split occured. The average allows for unbalanced splits to have less of an impact on our out-of-sample error estimation.

(n) Make two plots - one of your training data, and one for your test data - with the data scatterplotted and the decision boundary for your classifier. Comment on your decision boundary. Could it be improved?

Looking at our decision boundary, we can see that the boundary is not perfect. However, given our approach there is little improvement we could make. In order for us to improve our decision boundary we would need to allow from nonlinear boundaries to be drawn.

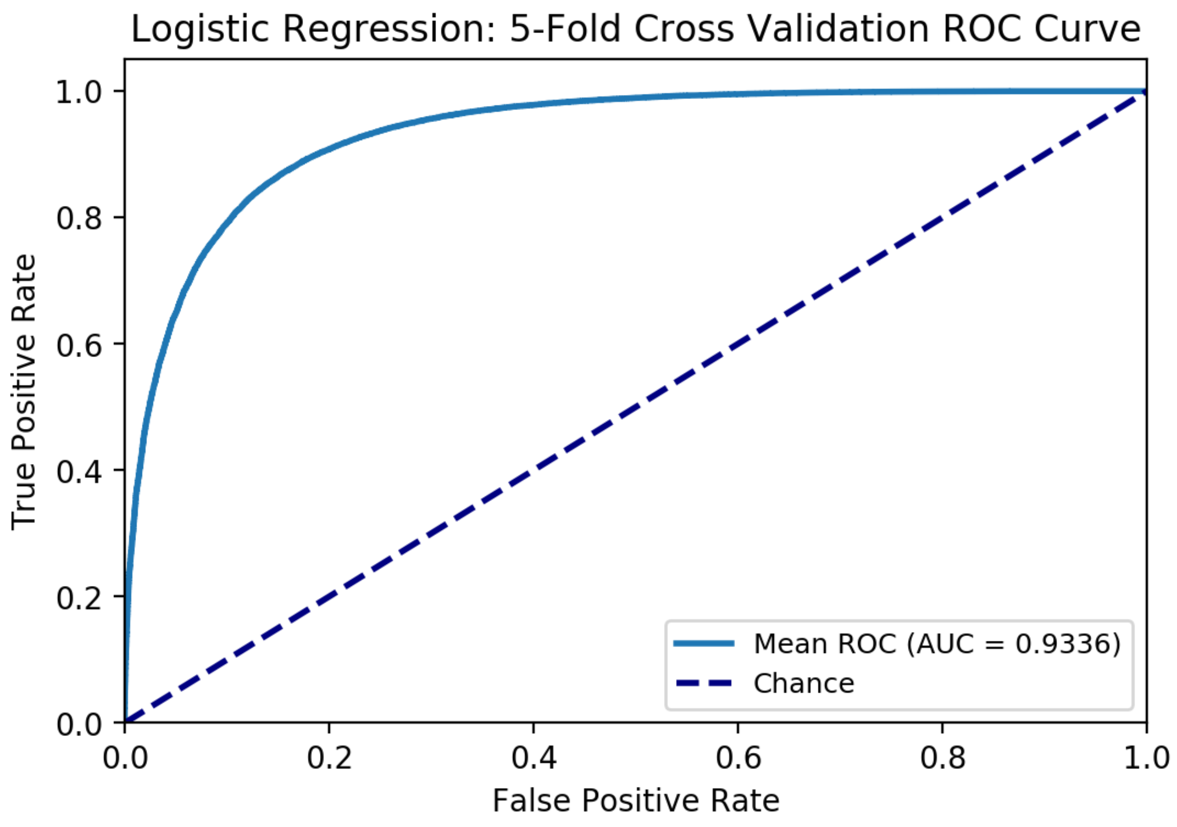
In [198]:
```
# Create meshgrid
x1 = np.linspace(-6, 6, 300)
x2 = np.linspace(-6, 6, 300)
xx, yy = np.meshgrid(x1, x2)
X_mesh = np.column_stack((np.ones(len(xx.ravel())),
                          xx.ravel(),yy.ravel()))

# Fit Logistic Regression
log_reg = LogisticReg()
log_reg.fit(x_train, y_train, x_test, y_test, learning_rate = 3)

# Prediction across the entire space
Z = log_reg.predict(X_mesh)
# Create binary choice between classes as opposed to probabilities
# Threshold = .5
Z = (Z > .5)*1

fig, axs = plt.subplots(1, 2, sharey = True, figsize=(12, 4))
fig.set_dpi(200)
fig.set_figheight(5)
fig.set_figwidth(10)

# Training dataset
axs[0].contourf(xx, yy, Z.reshape(xx.shape),
                cmap='PRGn', alpha = .3)
axs[0].scatter(x = x_train[y_train == 1,1], y = x_train[y_train == 1,2],
               c='green', edgecolors='w', label = 'Class 1')
axs[0].scatter(x = x_train[y_train == 0,1], y = x_train[y_train == 0,2],
               c='purple', edgecolors='w', label = 'Class 0')

# Test dataset
axs[1].contourf(xx, yy, Z.reshape(xx.shape),
                     cmap ='PRGn', alpha = .3)
axs[1].scatter(x = x_test[y_test == 1,1], y = x_test[y_test == 1,2],
               c='green', edgecolors='w')
axs[1].scatter(x = x_test[y_test == 0,1], y = x_test[y_test == 0,2],
               c='purple', edgecolors='w')

# Format plot
axs[0].set_xlim(xx.min(), yy.max())
axs[0].set_ylim(xx.min(), yy.max())
axs[0].set_xlabel("x1")
axs[0].set_ylabel("x2")
axs[0].set_title("Training Dataset")
axs[1].set_xlim(xx.min(), yy.max())
axs[1].set_ylim(xx.min(), yy.max())
axs[1].set_xlabel("x1")
axs[1].set_title("Test Dataset")
plt.suptitle('Logistic Regression Decision Boundary', fontsize = 16)
plt.tight_layout(rect=[0, 0.03, 1, .90])
fig.legend()
fig.show()
```



(o) Compare your trained model to random guessing. Show the ROC curve for your model and plot the chance diagonal. What area under the curve (AUC) does your model achieve? How does your model compare in terms of performance?

Our model achieves an AUC of 0.9336. Compared to random guessing at represented at an AUC of .5, our model preforms significantly better.

In [11]:
```
plt.figure(dpi = 200)
# Plot mean ROC from CV in part L
plt.plot(fpr, tpr, lw = 2,
             label='Mean ROC (AUC = {0:0.4f})'.format(roc_auc))

# Format ROC Plot
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
         label = 'Chance')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right", fontsize = 9)
plt.title('Logistic Regression: {}-Fold Cross Validation ROC Curve'.form
at(i))
plt.show()
```



2

[20 points] Digits classification

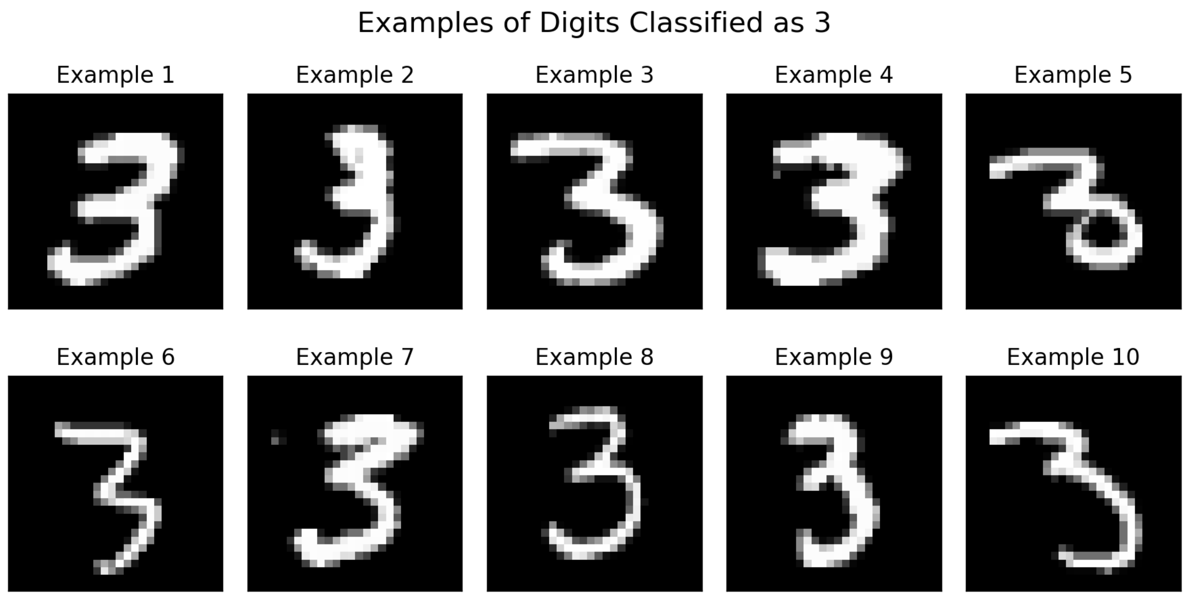
(a) Construct your dataset from the MNIST dataset (http://yann.lecun.com/exdb/mnist/) of handwritten digits, which has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image.

Your goal is to determine whether or not an example is a 3, therefore your binary classifier will seek to estimate $y = 1$ if the digit is a 3, and $y = 0$ otherwise. Create your dataset by transforming your labels into a binary format.

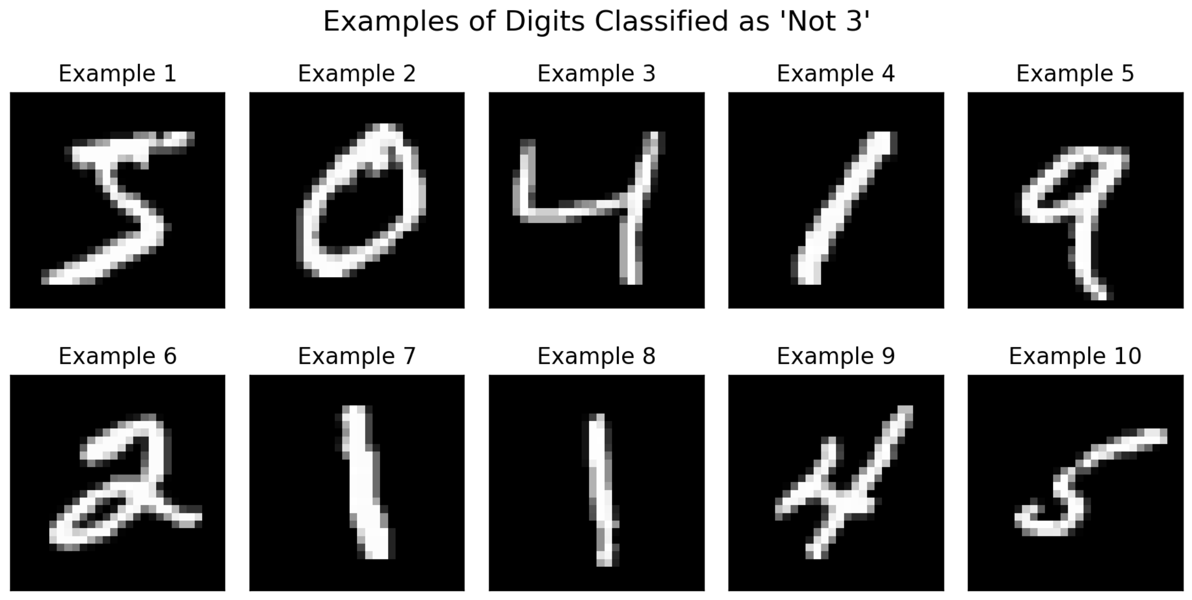
In [12]:
```
import matplotlib.pyplot as plt
import numpy as np
import random as rand
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import mean_squared_error, roc_curve, auc
from sklearn.metrics import precision_recall_curve, average_precision_sc
ore
import warnings
warnings.filterwarnings('ignore')

# Import MNIST dataset
from mlxtend.data import loadlocal_mnist
X_train, y_train = loadlocal_mnist(
       images_path='./Assignment Code/Assignment 3/MNIST/train-images-id
x3-ubyte',
       labels_path='./Assignment Code/Assignment 3/MNIST/train-labels-id
x1-ubyte')

X_test, y_test = loadlocal_mnist(
       images_path='./Assignment Code/Assignment 3/MNIST/t10k-images-idx
3-ubyte',
       labels_path='./Assignment Code/Assignment 3/MNIST/t10k-labels-idx
1-ubyte')

# Reshape for a parts A-D
x_train = np.reshape(X_train, (60000,28,28))
x_test = np.reshape(X_test, (10000,28,28))

# Create two classes, 3 or Not3
y_train = (y_train == 3)*1
y_test = (y_test == 3)*1
```

(b) Plot 10 examples of each class 0 and 1, from the training dataset.

In [13]:
```
# Create set of indexes for class 3 and Not3
train_3_index = np.where(y_train == 1)
train_not3_index = np.where(y_train == 0)

# Plot 10 examples of class 3
# Creating subplots
fig, axs = plt.subplots(2, 5)
fig.set_dpi(200)
fig.set_figwidth(9)
fig.set_figheight(5)

# For the first 10 class 3 images add them to the subplots
count = 0
for i in range(2):
    for j in range(5):
        image = x_train[train_3_index[0][count]]
        axs[i,j].imshow(image, cmap = 'gray')
        axs[i,j].set_xticks([], [])
        axs[i,j].set_yticks([], [])
        axs[i,j].title.set_text('Example ' + str(count+1))
        count += 1
        pass
    pass
plt.tight_layout(rect=[0, 0.03, 1, .95])
fig.suptitle('Examples of Digits Classified as 3',
            fontsize = 15)
plt.show()
```



In [201]:
```
# Plot 10 examples of class Not3
# Creating subplots
fig, axs = plt.subplots(2, 5)
fig.set_dpi(200)
fig.set_figwidth(9)
fig.set_figheight(5)


# For the first 10 class Not3 images add them to the subplots
count = 0
for i in range(2):
    for j in range(5):
        image = x_train[train_not3_index[0][count]]
        axs[i,j].imshow(image, cmap = 'gray')
        axs[i,j].set_xticks([], [])
        axs[i,j].set_yticks([], [])
        axs[i,j].title.set_text('Example ' + str(count+1))
        count += 1
        pass
    pass
plt.tight_layout(rect=[0, 0.03, 1, .95])
fig.suptitle('Examples of Digits Classified as \'Not 3\'',
             fontsize = 15)
plt.show()
```



(c) How many examples are present in each class? Are the classes balanced? What issues might this cause?

In [14]:
```
# How large are each class
print('Training sample size: ', len(y_train))
print('Number of class 3 in training sample: ', sum(y_train),
      '(', round(sum(y_train)/len(y_train)*100,1), '%)')

print('Number of class Not 3 in training sample: ',
      len(y_train) - sum(y_train),
      '(', round((1-sum(y_train)/len(y_train))*100,1), '%)')
```

```
Training sample size:  60000
Number of class 3 in training sample:  6131 ( 10.2 %)
Number of class Not 3 in training sample:  53869 ( 89.8 %)
```

Unfortunately, our classes are rather imbalanced. This makes sense given that 3 is just one of 10 digits with a $P(3) = 1/10$ and $P(Not3) = 1 - P(3) = 9/10$. Our class distribution is very close to this long-run frequency.

That said, class imbalance can become an issue in classification because when our model "decides" the best way to generate accurate predictions is to always predict the majority class. For example, in our case if we classified every training case as 'Not 3' we would have an training error rate of only 10% which, in other circumstances could be considered quite good.

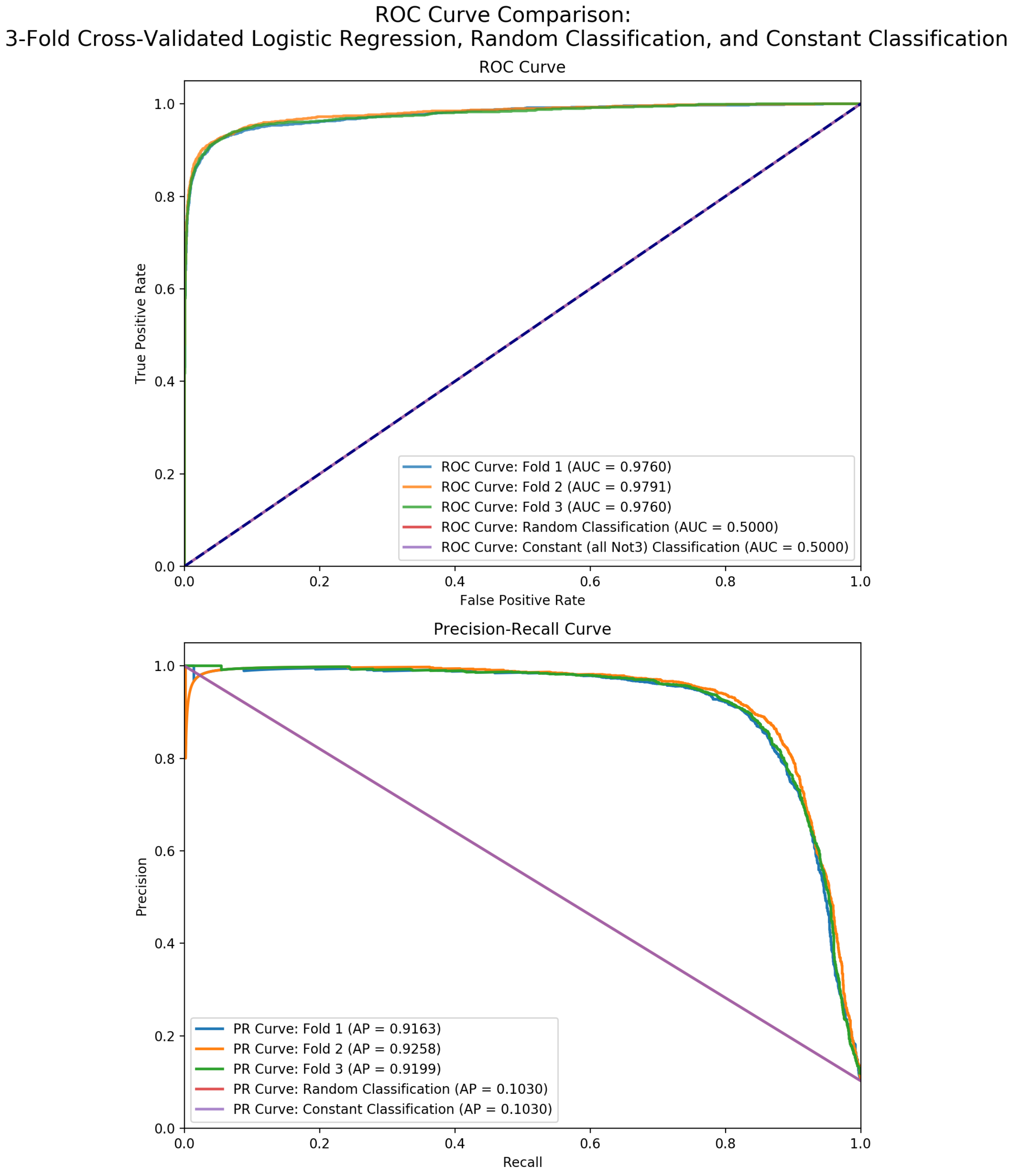
(d) Using cross-validation, train and test a classifier. Compare your performance against (1) a classifier that randomly guesses the class, and (2) a classifier that guesses that all examples are NOT 3's. Plot corresponding ROC curves and precision-recall curves. Describe the algorithm's performance and explain any discrepancies you find.

In terms of training and testing a classifier, I chose to fit a logistic classifier. Comparing the ROC and precision-recall curves between the folds of the classifier to a classifier that randomly guesses the class and a classifier that guess that all examples are not 3s showing some interesting trends.

We can see that each fold of cross-validated logistic classifier has an AUC of approximately .97 and an average precision of approximately .91. This is a successful classification, significantly outperforming random chance our single classifier. Because our preformance is so high, there is potential for concern of over-fitting. As we move forward in the problem and our analysis, we will adjust the parameters of our model and further examine the performance of our model.

In [15]:
```
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

fig, axs = plt.subplots(2,1)
fig.set_dpi(200)
fig.set_figwidth(8)
fig.set_figheight(14)

# Cross validation
cross_val = StratifiedKFold(n_splits=3)
# Loop through each fold
i = 0
for train, test in cross_val.split(X_train, y_train):
    # Fit logistic model on told
    logreg = LogisticRegression()
    fit = logreg.fit(X_train[train], y_train[train])
    # Calculate probablities of for each class
    y_probs = fit.predict_proba(X_train[test])

    # ROC Curve
    fpr, tpr, thresholds = roc_curve(y_train[test], y_probs[:, 1])
    roc_auc = auc(fpr, tpr)
    axs[0].plot(fpr, tpr, lw=2, alpha=0.8,
             label='ROC Curve: Fold {0} (AUC = {1:0.4f})'.format(i+1,
                                                                  roc_auc
))

    # PR Curve
    precision, recall, thresholds = precision_recall_curve(y_train[test
],
                                                              y_probs[:, 1
])
    average_precision = average_precision_score(y_train[test],
                                                y_probs[:, 1])
    axs[1].plot(recall, precision, lw = 2,
       label='PR Curve: Fold {0} (AP = {1:0.4f})'.format(i+1,
                                                         average_precisi
on))
    i += 1
    pass

# Random Classifier
# Fit random classifier
dummy = DummyClassifier(strategy = 'uniform')
fit = dummy.fit(X_train[:50000,:], y_train[:50000])
y_probs = fit.predict_proba(X_train[50000:,:])

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_train[50000:], y_probs[:, 1])
roc_auc = auc(fpr, tpr)
axs[0].plot(fpr, tpr, lw=2, alpha=0.8,
   label='ROC Curve: Random Classification (AUC = {0:0.4f})'.format(roc_
auc))
# PR Curve
precision, recall, thresholds = precision_recall_curve(y_train[50000:],
                                                       y_probs[:, 1])
```

```
average_precision = average_precision_score(y_train[50000:],
                                            y_probs[:, 1])
axs[1].plot(recall, precision, lw = 2, alpha = .8,
   label='PR Curve: Random Classification (AP = {0:0.4f})'.format(averag
e_precision))



# Constant 0 (Not3) Classifier
# Fit constant classifier
dummy = DummyClassifier(strategy = 'constant', constant = 0)
fit = dummy.fit(X_train[:50000,:], y_train[:50000])
y_probs = fit.predict_proba(X_train[50000:,:])

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_train[50000:], y_probs[:, 1])
roc_auc = auc(fpr, tpr)
axs[0].plot(fpr, tpr, lw=2, alpha=0.8,
   label='ROC Curve: Constant (all Not3) Classification (AUC = {0:0.4f}
)'.format(roc_auc))
# PR Curve
precision, recall, thresholds = precision_recall_curve(y_train[50000:],
                                                       y_probs[:, 1])
average_precision = average_precision_score(y_train[50000:],
                                            y_probs[:, 1])
axs[1].plot(recall, precision, lw = 2, alpha = .8,
   label='PR Curve: Constant Classification (AP = {0:0.4f})'.format(aver
age_precision))


# Format ROC Plot
axs[0].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axs[0].set_xlim([0.0, 1.0])
axs[0].set_ylim([0.0, 1.05])
axs[0].set_xlabel('False Positive Rate')
axs[0].set_ylabel('True Positive Rate')
axs[0].legend(loc="lower right")
axs[0].set_title('ROC Curve')

# Format PR Plot
axs[1].set_xlim([0.0, 1.0])
axs[1].set_ylim([0.0, 1.05])
axs[1].set_xlabel('Recall')
axs[1].set_ylabel('Precision')
axs[1].legend(loc="lower left")
axs[1].set_title('Precision-Recall Curve')

# Add overall title
plt.tight_layout(rect=[0, 0.1, 1, 0.95])
plt.suptitle('ROC Curve Comparison:\n {0}-Fold Cross-Validated Logistic
 Regression, Random Classification, and Constant Classification'.format(
i),
             fontsize = 16)
plt.show()
```

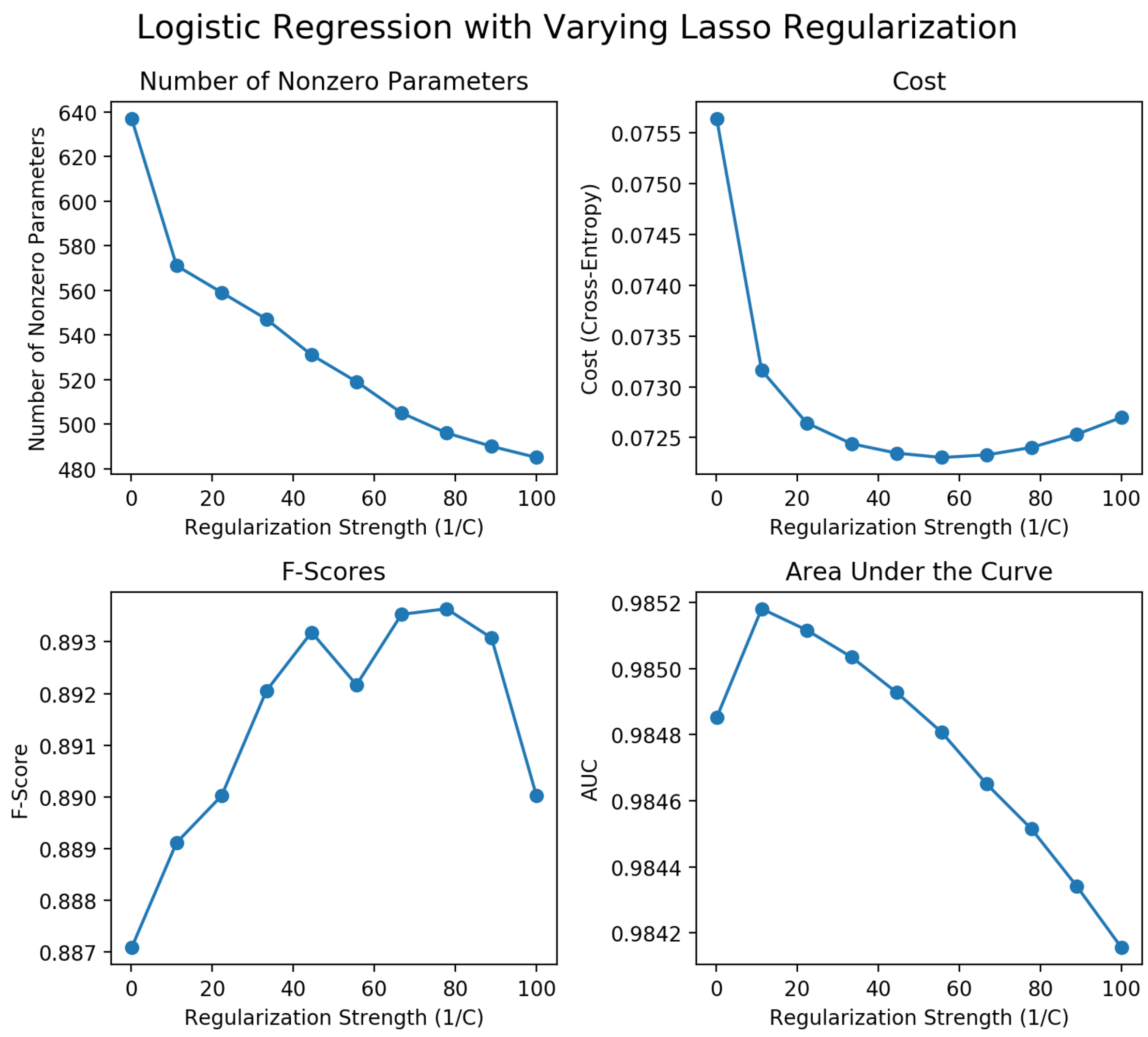


(f) Using a logistic regression classifier (a linear classifier), apply lasso regularization and retrain the model and evaluate its performance over a range of values on the regularization coefficient. You can implement this using the LogisticRegression (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) module (DO NOT use your function from question 1) and activating the 'l1' penalty; the parameter C is the inverse of the regularization strength. As you vary the regularization coefficient, plot (1) the number of model parameters that are estimated to be nonzero; (2) the logistic regression cost function, which you created a function to evaluate in the Question 1; (3) F_1-score, and (4) area under the curve (AUC). Describe the implications of your findings.

We see several trends in our plots. First, we see that with increased regularization the number of non-zero parameters decreases. This is to be expected, as it is the definition of lasso regularization. Second, we see the cost function decrease and eventually increase. Conversely, we see our F1-score increase and eventually decrease. Finally, we very slight changes in our AUC with a slight increase before quickly decreasing. Overall, this findings suggest selecting a level of regularization perhaps around 40 or C = 1/40. Ultimately, it should be noted that performance changes in the F1-score and the AUC are very slight and any shift in regularization appears to have only small effects.

In [16]:
```
from sklearn.metrics import f1_score

# Create lists of
#   1. Number of model parameters that are estimated to be nonzero
#   2. Logistic regression cost function (from 1)
#   3. F-score
#   4. AUC score
Cs = []
params = []
costs = []
F_scores = []
auc_scores = []

def costCalc(Y, Y_hat):
    N = len(Y)
    cost = - Y.T.dot(np.log(Y_hat)) - (1-Y).T.dot(np.log(1- Y_hat))
    return cost / N

# Iterate over an evenly spaced list of inverse regularization
c_list = np.linspace(.1,100, 10)
c_list = np.divide(1, c_list)

for c in c_list:
    Cs.append(c)

    # Fit a logistic regression
    # Using penalty = 'l1'
    log_reg = LogisticRegression(penalty = 'l1', C = c)
    fit = log_reg.fit(X_train, y_train)
    y_probs = fit.predict_proba(X_test)
    y_preds = fit.predict(X_test)

    # Number of model parameters that are estimated to be nonzero
    non_zero_coefs = sum(fit.coef_[0,] != 0)
    params.append(non_zero_coefs)

    # Costs
    cost = costCalc(y_test, y_probs[:, 1])
    costs.append(cost)

    # F-score
    f_score = f1_score(y_test, y_preds)
    F_scores.append(f_score)

    # AUC
    fpr, tpr, thresholds = roc_curve(y_test[:,], y_probs[:, 1])
    roc_auc = auc(fpr, tpr)
    auc_scores.append(roc_auc)
    pass
```

In [17]:
```
# Plot the changes in metrics
fig, axs = plt.subplots(2,2)
fig.set_dpi(200)
fig.set_figwidth(8)
fig.set_figheight(8)

# Number of nonzero parameters
axs[0,0].plot(np.divide(1, Cs), params, '-o')
axs[0,0].set_title('Number of Nonzero Parameters')
#axs[0,0].set_xlim([0, 1.0])
axs[0,0].set_xlabel('Regularization Strength (1/C)')
axs[0,0].set_ylabel('Number of Nonzero Parameters')

# Cost function
axs[0,1].plot(np.divide(1, Cs), costs, '-o')
axs[0,1].set_title('Cost')
#axs[0,1].set_xlim([0, 1.0])
axs[0,1].set_xlabel('Regularization Strength (1/C)')
axs[0,1].set_ylabel('Cost (Cross-Entropy)')

# F-scores
axs[1,0].plot(np.divide(1, Cs), F_scores, '-o')
axs[1,0].set_title('F-Scores')
#axs[1,0].set_xlim([0, 1.0])
axs[1,0].set_xlabel('Regularization Strength (1/C)')
axs[1,0].set_ylabel('F-Score')

# AUC curve
axs[1,1].plot(np.divide(1, Cs), auc_scores, '-o')
axs[1,1].set_title('Area Under the Curve')
#axs[1,1].set_xlim([0, 1.0])
#axs[1,1].set_ylim([0.9849, 0.9853])
axs[1,1].set_xlabel('Regularization Strength (1/C)')
axs[1,1].set_ylabel('AUC')

# Add overall title
plt.tight_layout(rect=[0, 0.1, 1, 0.95])
plt.suptitle('Logistic Regression with Varying Lasso Regularization\n',
             fontsize = 16)
plt.show()
```



3

[40 points] Supervised learning exploration

For this exercise, you will construct and implement a supervised learning problem solution/experiment. Describe your process and answer these questions clearly and thoroughly. Part of the grade in this assignment is devoted to the quality and professionalism of your work.

(a) Identify a question or problem that's of interest to you and that could be addressed using classification or regression. Explain why it's interesting and what you'd like to accomplish. This should exhibit creativity, and you are not allowed to use the Iris dataset, the Kaggle Titanic dataset, or the Kaggle chocolate dataset.

The World Health Organization (WHO) estimates that each year approximately one million people die from suicide, which represents a global mortality rate of 16 people per 100,000 or one death every 40 seconds. The first step in suicide prevention is suicide detection. The following analysis examines global suicide rates and hopes to identify and predict above-average suicide rates among different cohorts globally, across the socio-economic spectrum.

The dataset used in our analysis can be found on Kaggle (https://www.kaggle.com/russellyates88/suicide-rates-overview-1985-to-2016/home) and is a compilation four other datasets linked by time and place, and was built to find signals correlated to increased suicide rates among different cohorts globally, across the socio-economic spectrum.

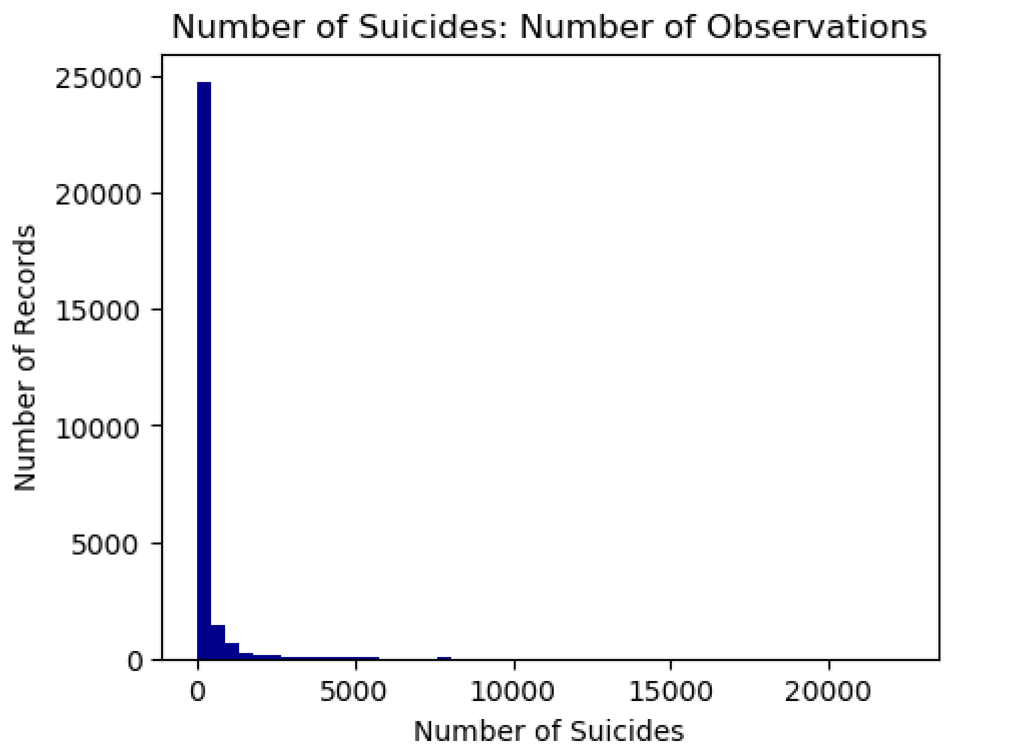
(b) Download the data and plot the data to describe it.

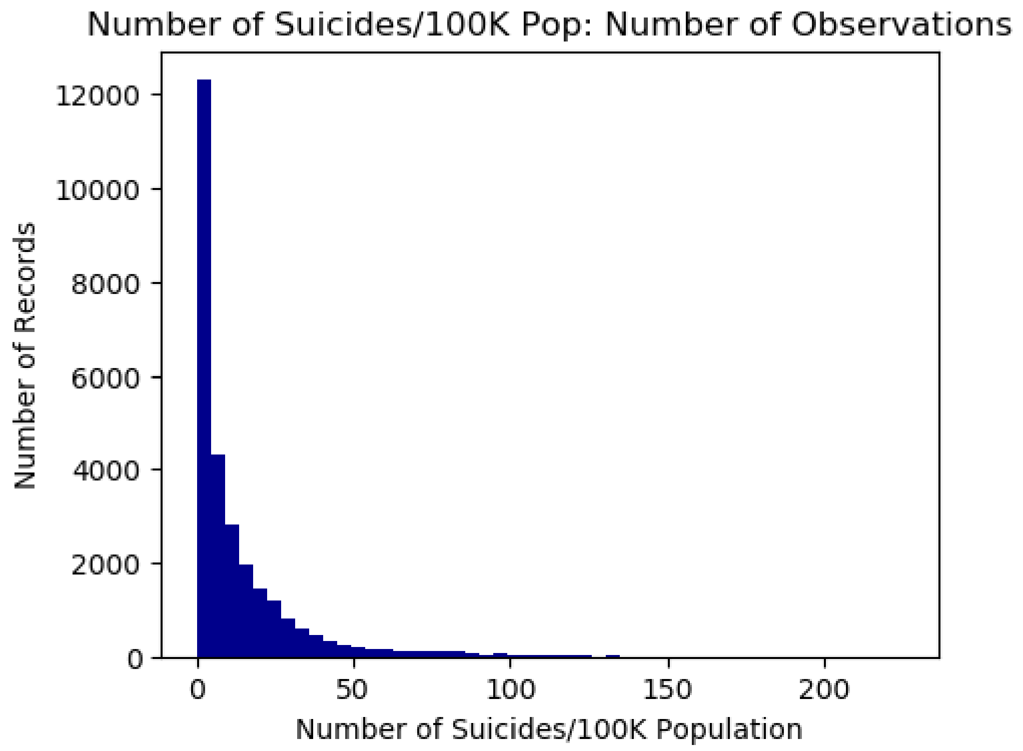
Our dataset has **27,820 obervations** and the following variables:

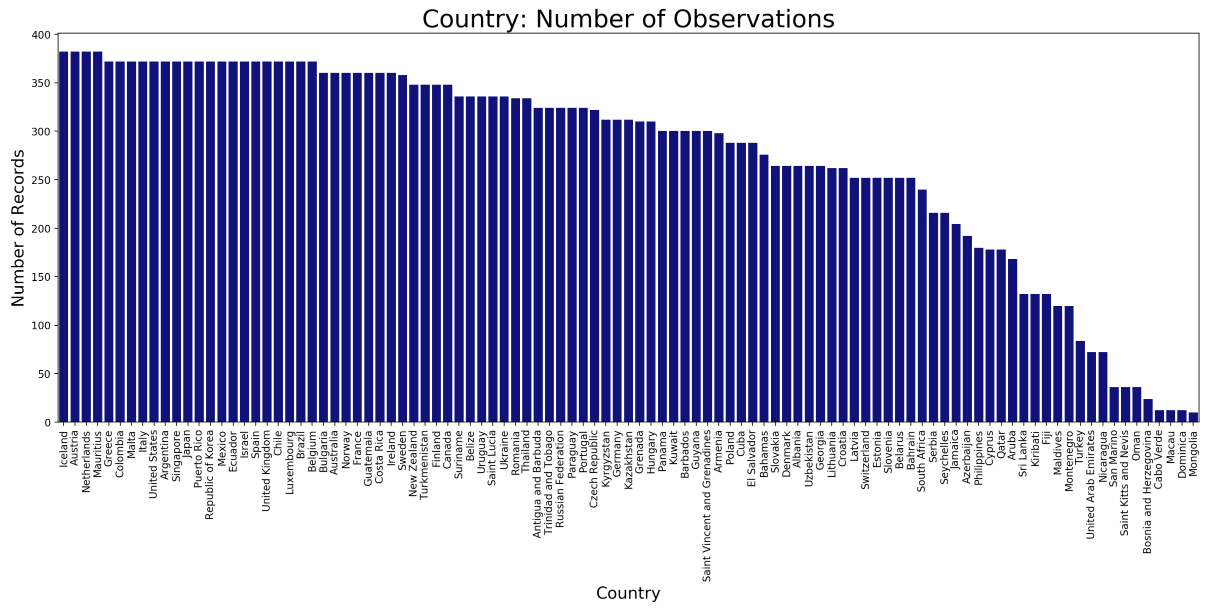
Variable	**Description**
country	Country name
year	Year of suicide (1985-2016)
sex	Male or female
age	Categorial age bracket
suicides_no	Number of suicides
population	Number of people in segment
suicides/100k pop	Suicides per 100,000 population
country-year	Combination of country and year
HDI for year	Human Development Index, higher scores are successes
gdp_for_year	Gross Domestic Product for the country and year
gdp_per_capita	Gross Domestic Product per capita for the country and year
generation	Categorial generation

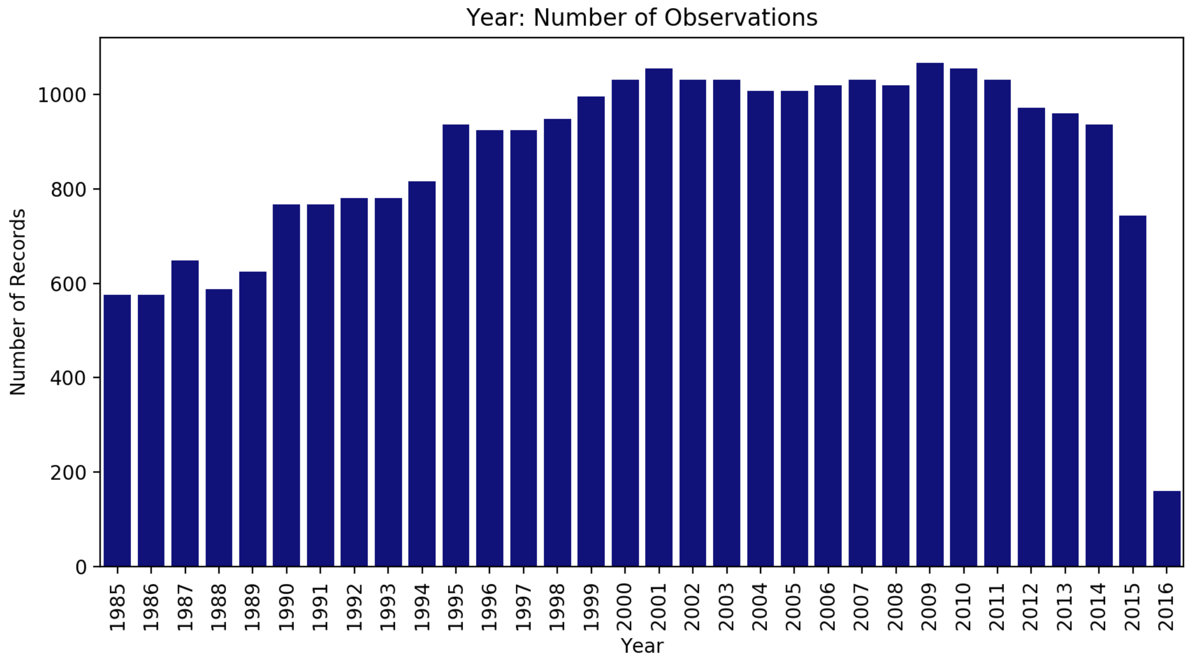
In [18]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Import the data
suicide = pd.read_csv('./Assignment Code/Assignment 3/Suicide/master.cs
v',
                      index_col=0,parse_dates=[0])
# Originally "Country" is the index column, this creates an index column
suicide = suicide.reset_index()
```

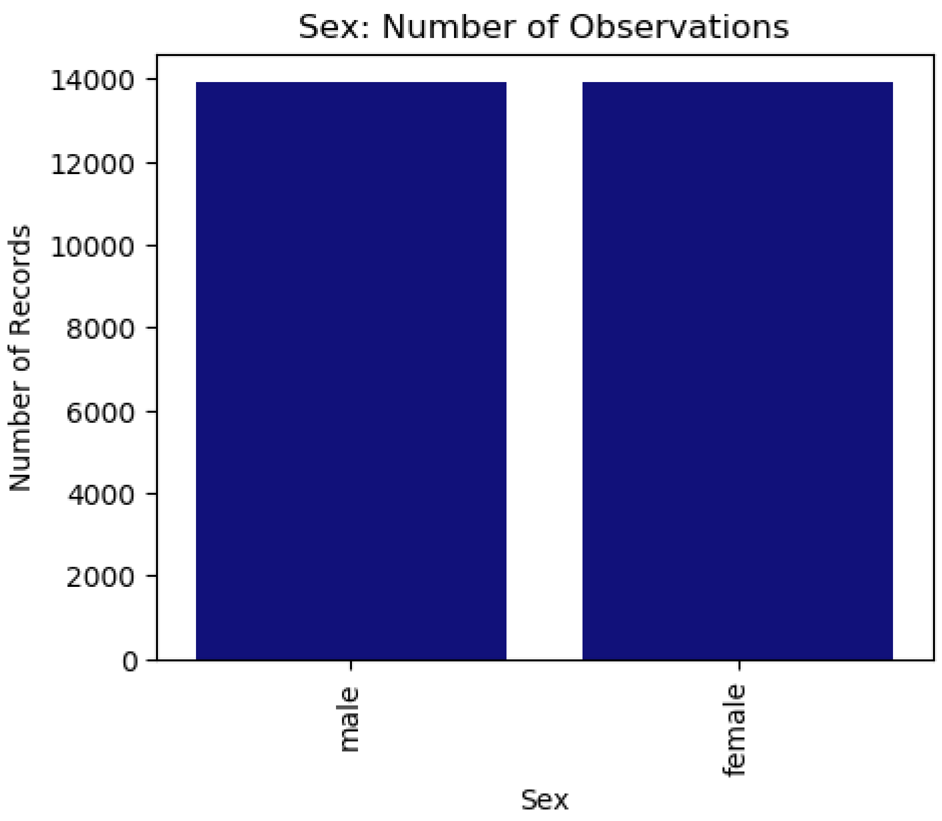
In [19]:
```
# Plot variables
# Potential Outcomes
# Suicides_no
plt.figure(dpi = 100, figsize=(5,4))
plt.hist(suicide['suicides_no'], color='darkblue', bins = 50)
plt.xlabel('Number of Suicides', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Number of Suicides: Number of Observations',fontsize = 12)
plt.show()

# Suicides/100k pop
plt.figure(dpi = 100, figsize=(5,4))
plt.hist(suicide['suicides/100k pop'], color='darkblue', bins = 50)
plt.xlabel('Number of Suicides/100K Population', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Number of Suicides/100K Pop: Number of Observations',
          fontsize = 12)
plt.show()



# Potential Predictors
# country
plt.figure(dpi = 200, figsize=(20,7))
fig = sns.countplot('country',data = suicide, color='darkblue',
                    order = suicide['country'].value_counts().index)
fig.set_xticklabels(fig.get_xticklabels(), rotation=90)
plt.xlabel('Country', fontsize = 16)
plt.ylabel('Number of Records', fontsize = 16)
plt.title('Country: Number of Observations',fontsize = 24)
plt.show()

# year
plt.figure(dpi = 200, figsize=(10,5))
fig = sns.countplot('year',data = suicide, color='darkblue',)
fig.set_xticklabels(fig.get_xticklabels(), rotation=90)
plt.xlabel('Year', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Year: Number of Observations',fontsize = 12)
plt.show()

# Sex
plt.figure(dpi = 100, figsize=(5,4))
fig = sns.countplot('sex',data = suicide, color='darkblue')
fig.set_xticklabels(fig.get_xticklabels(), rotation=90)
plt.xlabel('Sex', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Sex: Number of Observations',fontsize = 12)
plt.show()

# Age
plt.figure(dpi = 100, figsize=(5,4))
fig = sns.countplot('age',data = suicide, color='darkblue',
                   order = suicide['age'].value_counts().index)
fig.set_xticklabels(fig.get_xticklabels(), rotation=90)
plt.xlabel('Age', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Age: Number of Observations',fontsize = 12)
plt.show()

# Population
plt.figure(dpi = 100, figsize=(5,4))
plt.hist(suicide['population']/100000, color='darkblue', bins = 50)
plt.xlabel('Population (100K)', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Population: Number of Observations',fontsize = 12)
plt.show()

# HDI
plt.figure(dpi = 100, figsize=(5,4))
plt.hist(suicide[~np.isnan(suicide['HDI for year'])]['HDI for year'],
         color='darkblue', bins = 50)
plt.xlabel('HDI for Year', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('HDI for Year: Number of Observations',fontsize = 12)
plt.show()

# GDP per capita
plt.figure(dpi = 100, figsize=(5,4))
fig = sns.countplot('generation',data = suicide, color='darkblue',
                    order = suicide['generation'].value_counts().index)
fig.set_xticklabels(fig.get_xticklabels(), rotation=90)
plt.xlabel('Generation', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Generation: Number of Observations',fontsize = 12)
plt.show()
```

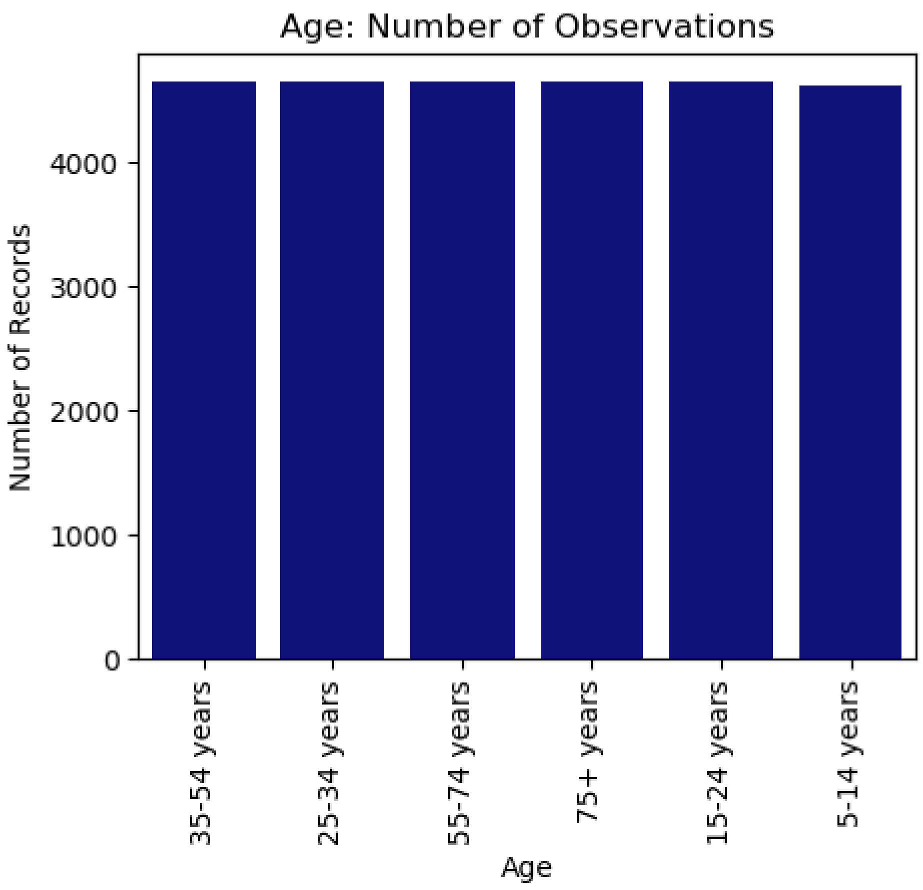

Number of Suicides: Number of Observations
Number of Records
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
Number of Suicides

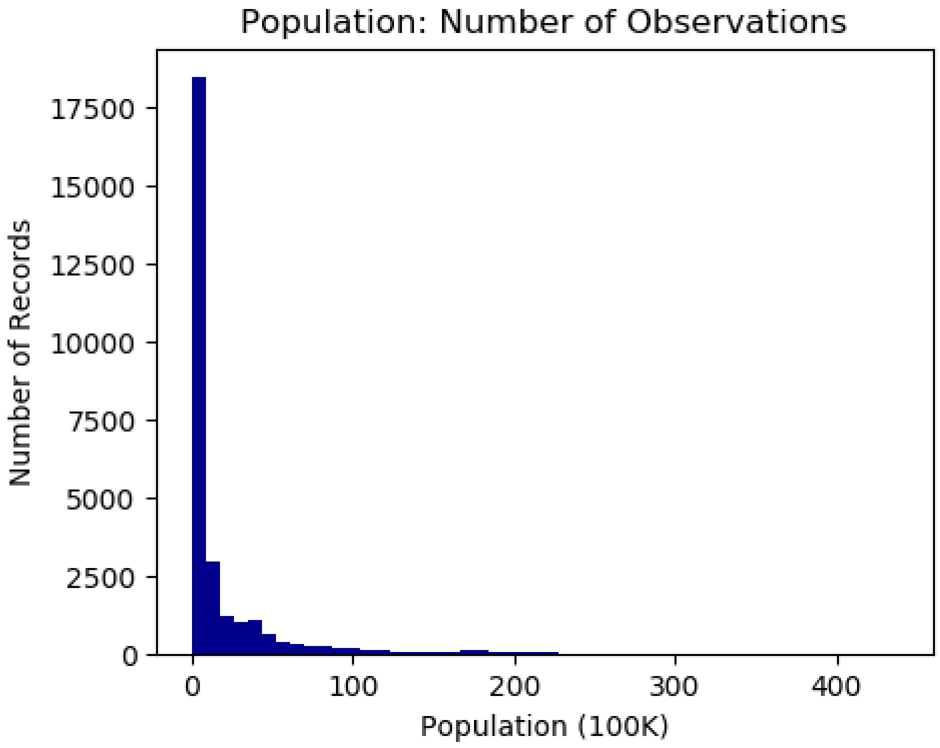

Number of Suicides/100K Pop: Number of Observations
Number of Records
12000
10000
8000
6000
4000
2000
0
0
50
100
150
200
Number of Suicides/100K Population

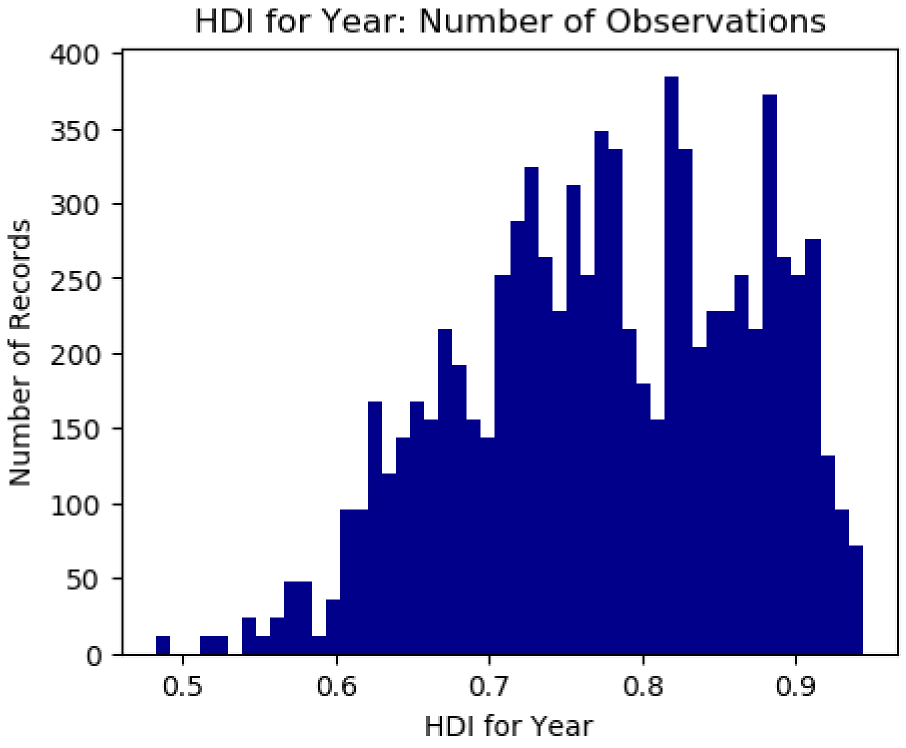

Country: Number of Observations
Number of Records
Country

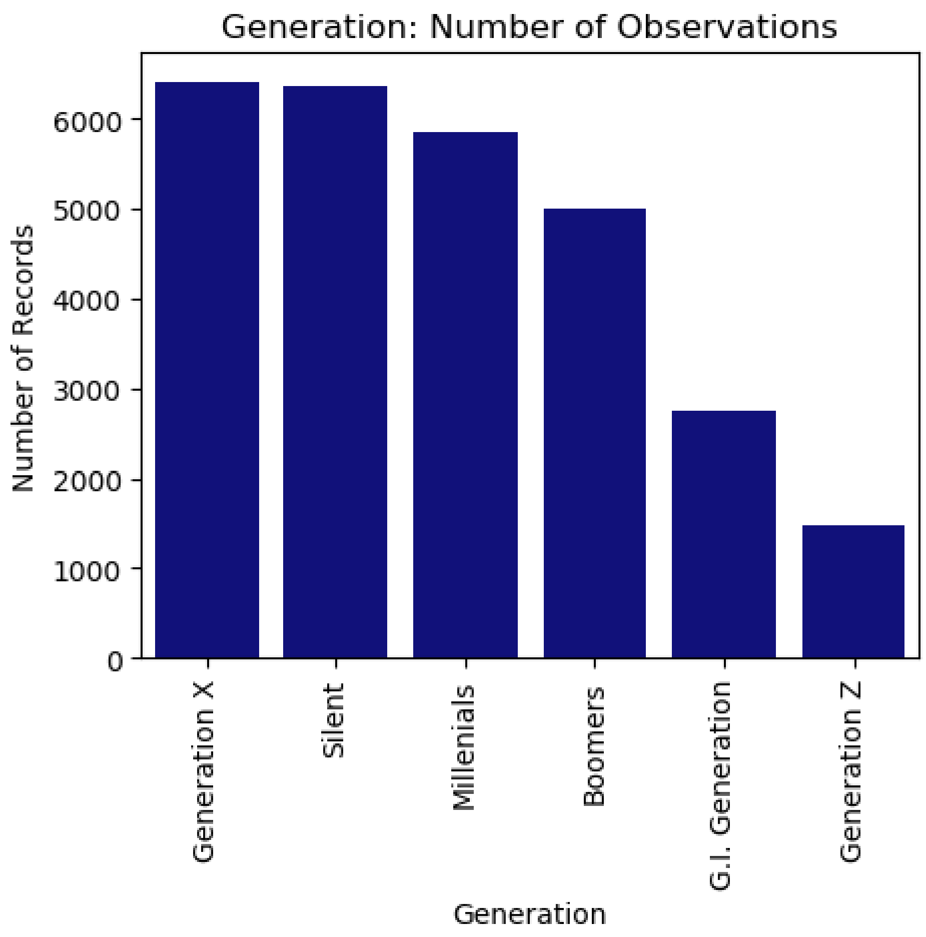

Year: Number of Observations
Number of Records
Year


Sex: Number of Observations
Number of Records
0
2000
4000
6000
8000
10000
12000
14000
male
female
Sex


Age: Number of Observations
Number of Records
0
1000
2000
3000
4000
35-54 years
25-34 years
55-74 years
75+ years
15-24 years
5-14 years
Age


Population: Number of Observations
Number of Records
17500
15000
12500
10000
7500
5000
2500
0
0
100
200
300
400
Population (100K)


HDI for Year: Number of Observations
Number of Records
400
350
300
250
200
150
100
50
0
0.5
0.6
0.7
0.8
0.9
HDI for Year



(c) Formulate your supervised learning question: (a) What is your target variable (what are you trying to predict) and what predictors do you have available? Does your dataset require any preprocessing: is it clean (no missing values or erroneous data) and normalized (are each of the predictors of the same magnitude)?

For this analysis, we'll be attempting to predict above-average suicide rates. According to WHO, in 2016 the global average suicide rate was 10.6 (10.6 suicides in 100K people). Therefore, we classify records with suicide rates above 10.6 to be above-average.

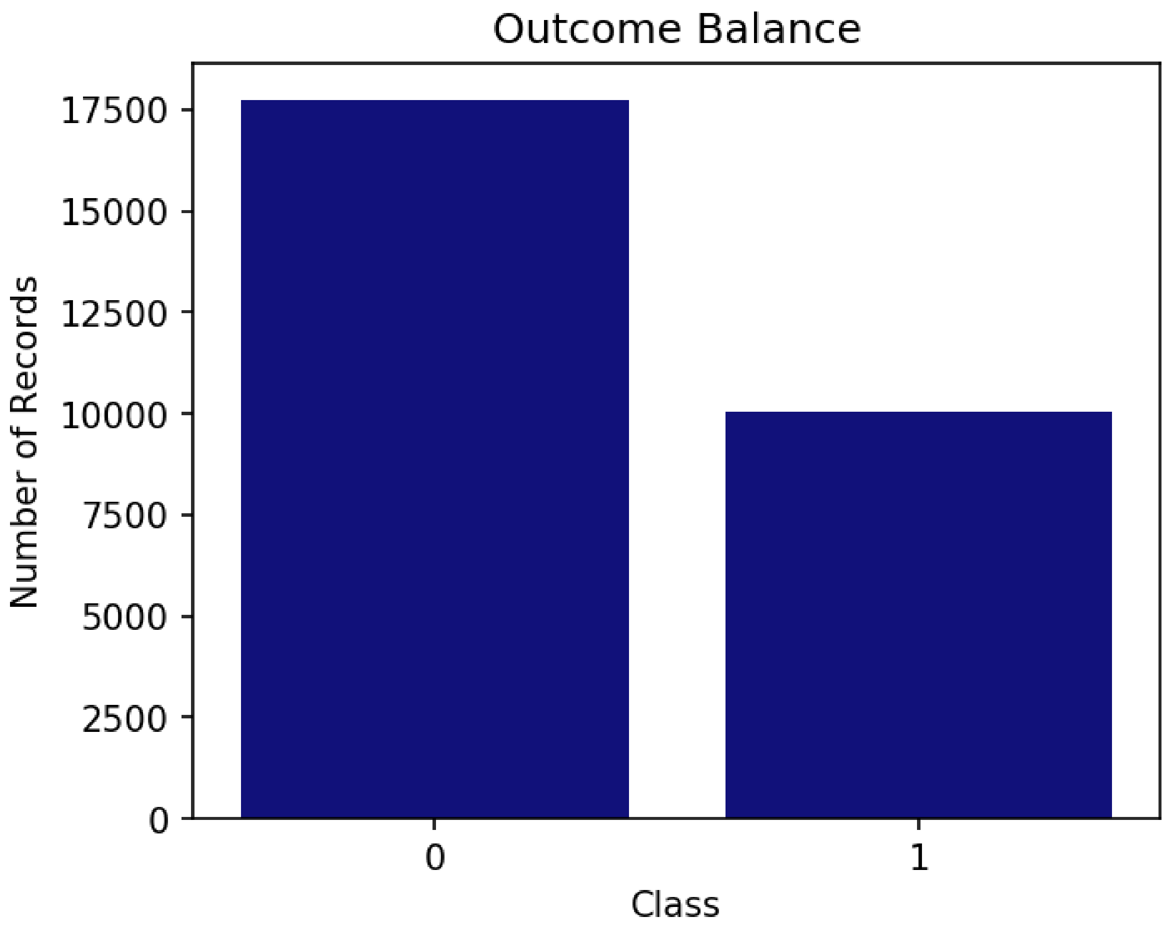
In terms of predictors, we will be using country, country population, gdp per capita, year, age, generation, and sex. We have removed HDI as a predictor because ~70% of the values were missing. We have also removed gdp for year, country-year, and suicides_no to prevent multicollinearity. In terms of transformations, we have logged population due to the high skew. We also normalized all continuous variables and one-hot encoded all categorical variables.

This dataset is maintained by the World Bank, and, from our analysis, does not appear to have any erroneous data.

In [20]:
```
# Create outcome variable based on US avg
y = (suicide['suicides/100k pop'] > 10.6)*1

# Check for missing observations
print('Proportion of Missing Values:\n',
      round(suicide.isna().sum()/len(suicide),3))

# Logged variables
suicide['population_log'] = np.log(suicide['population'])

# Removed duplicate variables (multicollinearity)
suicide = suicide.drop(['HDI for year', ' gdp_for_year ($) ', 'country-y
ear',
             'suicides_no', 'suicides/100k pop', 'population'], axis=1)

# One hot encoding for categorical values
suicide_cat_var = suicide[['sex', 'age', 'generation', 'country']]
one_hot = pd.get_dummies(suicide_cat_var)

x = np.column_stack([suicide['year'], suicide['population_log'],
              suicide['gdp_per_capita ($)'], one_hot])
```

```
Proportion of Missing Values:
 country                0.000
year                   0.000
sex                    0.000
age                    0.000
suicides_no            0.000
population             0.000
suicides/100k pop      0.000
country-year           0.000
HDI for year           0.699
 gdp_for_year ($)      0.000
gdp_per_capita ($)     0.000
generation             0.000
dtype: float64
```

In [21]:
```
# Outcome
plt.figure(dpi = 150, figsize=(5,4))
sns.countplot(x = y, color='darkblue')
plt.xlabel('Class', fontsize = 10)
plt.ylabel('Number of Records', fontsize = 10)
plt.title('Outcome Balance',fontsize = 12)
plt.show()
```



(d) What supervised learning technique will you use and why?

For our classifer we will use logisitic regression because of the binary nature of our problem. We will also use lasso regularization due to the large number of predictors created through one-hot encoding.

(e) How will you evaluate performance and know whether you succeeded (e.g. ROC curves for binary classification, mean square error or R^2 for regression)?

We will evaluate our model using ROC curves and prediction accuracy.

(f) Divide your dataset into training and testing datasets OR implement cross validation. Explain your approach and why you adopted it.

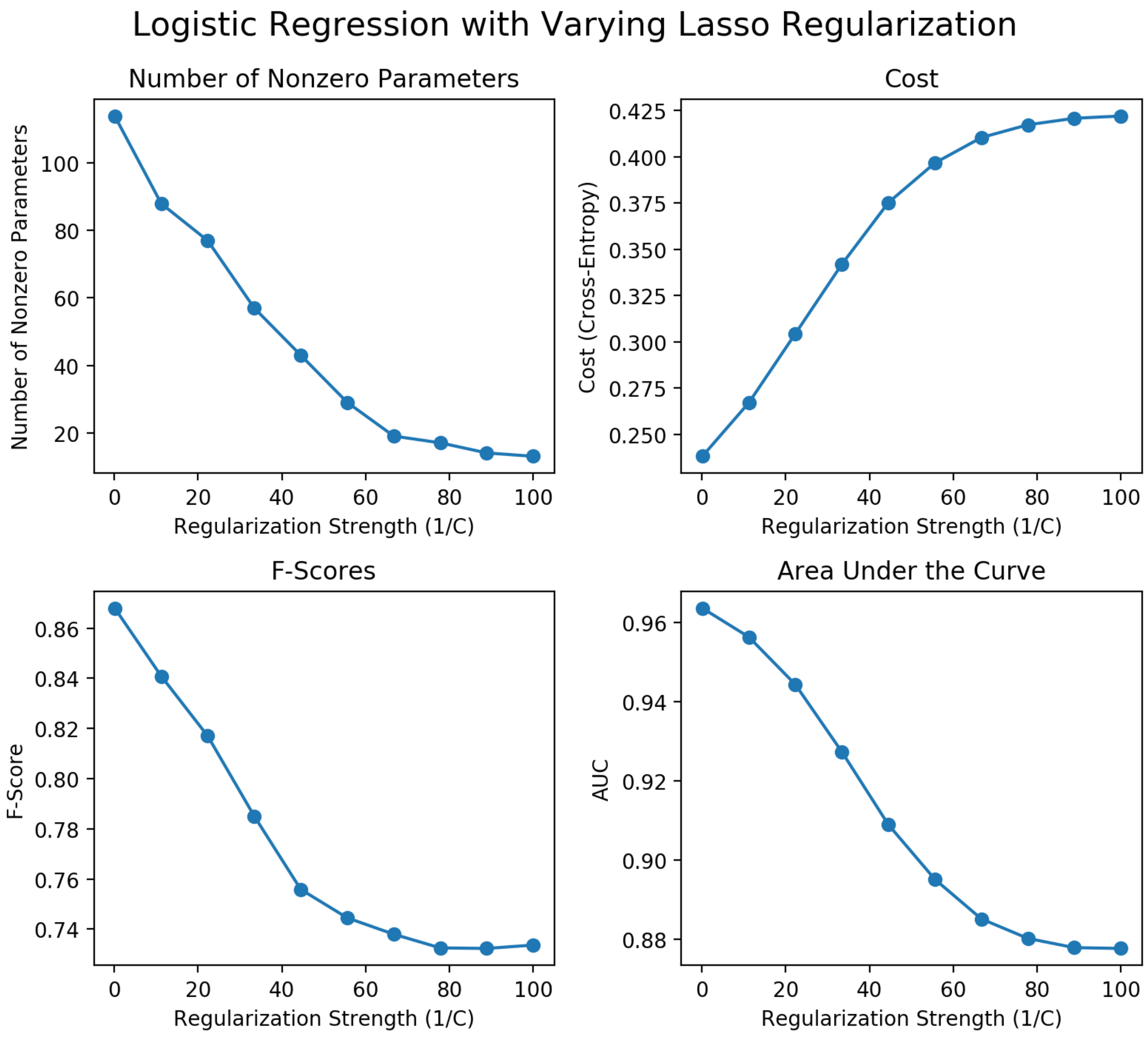
First, using a 70:30 train/test split we tested multiple values of the lasso regularization parameter to see which value optimized fit. However, what we found was the smallest amount of regularization produced the best fit. Moving forward we use an inverse regularization strength of C = .01.

Second, we used cross validation to examine our logistic regression fit. By using cross validation we are able to effectively create an predict out of sample error without relying on a single training/test split. The results of this analysis are below.

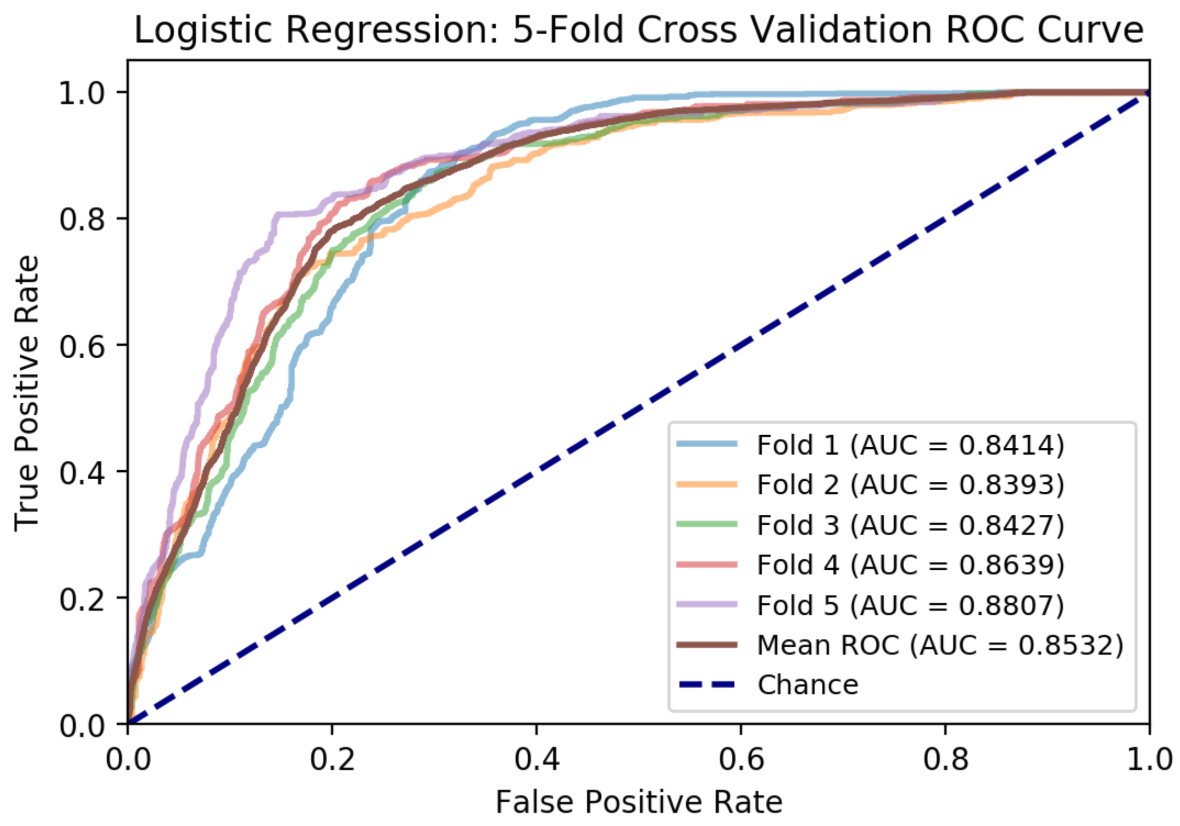
(g) Run your analysis and show your performance. Include plots of your data and of performance.

In [22]:
```
from sklearn.metrics import f1_score
import warnings
warnings.filterwarnings('ignore')

# Shuffle order order of dataset
N = len(y)
shuffle = np.arange(N)
np.random.shuffle(shuffle)
x_shuff = x[shuffle]
y_shuff = y[shuffle]

# Partition on a 70:30 split
split = .7
break_point = int(split*N)
x_train = x_shuff[:break_point]
y_train = y_shuff[:break_point]
x_test = x_shuff[break_point:]
y_test = y_shuff[break_point:]

# Create lists of
#   1. Number of model parameters that are estimated to be nonzero
#   2. Logistic regression cost function (from 1)
#   3. F-score
#   4. AUC score
Cs = []
params = []
costs = []
F_scores = []
auc_scores = []

def costCalc(Y, Y_hat):
    N = len(Y)
    cost = - Y.T.dot(np.log(Y_hat)) - (1-Y).T.dot(np.log(1- Y_hat))
    return cost / N

# Iterate over an evenly spaced list of inverse regularization
# coefficents
c_list = np.linspace(.1,100, 10)
c_list = np.divide(1, c_list)

for c in c_list:
    Cs.append(c)

    # Fit a logistic regression
    # Using penalty = 'l1'
    log_reg = LogisticRegression(penalty = 'l1', C = c)
    fit = log_reg.fit(x_train, y_train)
    y_probs = fit.predict_proba(x_test)
    y_preds = fit.predict(x_test)

    # Number of model parameters that are estimated to be nonzero
    non_zero_coefs = sum(fit.coef_[0,] != 0)
    params.append(non_zero_coefs)

    # Costs
    cost = costCalc(y_test, y_probs[:, 1])
    costs.append(cost)

    # F-score
    f_score = f1_score(y_test, y_preds)
    F_scores.append(f_score)

    # AUC
    fpr, tpr, thresholds = roc_curve(y_test[:,], y_probs[:, 1])
    roc_auc = auc(fpr, tpr)
    auc_scores.append(roc_auc)
    pass
```

In [23]:
```
# Plot the changes in metrics
fig, axs = plt.subplots(2,2)
fig.set_dpi(200)
fig.set_figwidth(8)
fig.set_figheight(8)

# Number of nonzero parameters
axs[0,0].plot(np.divide(1, Cs), params, '-o')
axs[0,0].set_title('Number of Nonzero Parameters')
#axs[0,0].set_xlim([0, 1.0])
axs[0,0].set_xlabel('Regularization Strength (1/C)')
axs[0,0].set_ylabel('Number of Nonzero Parameters')

# Cost function
axs[0,1].plot(np.divide(1, Cs), costs, '-o')
axs[0,1].set_title('Cost')
#axs[0,1].set_xlim([0, 1.0])
axs[0,1].set_xlabel('Regularization Strength (1/C)')
axs[0,1].set_ylabel('Cost (Cross-Entropy)')

# F-scores
axs[1,0].plot(np.divide(1, Cs), F_scores, '-o')
axs[1,0].set_title('F-Scores')
#axs[1,0].set_xlim([0, 1.0])
axs[1,0].set_xlabel('Regularization Strength (1/C)')
axs[1,0].set_ylabel('F-Score')

# AUC curve
axs[1,1].plot(np.divide(1, Cs), auc_scores, '-o')
axs[1,1].set_title('Area Under the Curve')
#axs[1,1].set_xlim([0, 1.0])
#axs[1,1].set_ylim([0.9849, 0.9853])
axs[1,1].set_xlabel('Regularization Strength (1/C)')
axs[1,1].set_ylabel('AUC')

# Add overall title
plt.tight_layout(rect=[0, 0.1, 1, 0.95])
plt.suptitle('Logistic Regression with Varying Lasso Regularization\n',
             fontsize = 16)
plt.show()
```


Logistic Regression with Varying Lasso Regularization
Number of Nonzero Parameters
Number of Nonzero Parameters
Regularization Strength (1/C)
Cost
Cost (Cross-Entropy)
Regularization Strength (1/C)
F-Scores
F-Score
Regularization Strength (1/C)
Area Under the Curve
AUC
Regularization Strength (1/C)

In [24]:
```
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import mean_squared_error, roc_curve, auc
from sklearn.metrics import accuracy_score


# Normalizing function for numerical variables
# Note: Does not infuse test dataset with info about training
def normalize(x_train, x_test):
    # Calculate min and max of training dataset
    mini = np.min(x_train)
    maxi = np.max(x_train)

    # Normalize training and test sets
    x_train_norm = (x_train - mini) / (maxi - mini)
    x_test_norm = (x_test - mini) / (maxi - mini)

    return x_train_norm, x_test_norm


# Cross validation
cross_val = StratifiedKFold(n_splits=5)
y_test_all = []
y_probs_all = []
acc_all = []

# Loop through each fold
plt.figure(dpi = 200)
i = 0
for train, test in cross_val.split(x, y):
    # Create fold training and test sets
    x_train, x_test = x[train], x[test]
    y_train, y_test = y[train], y[test]

    # Normalize continuous variables
    # All variables after first 3 are one-hot encoded
    for j in range(3):
        x_train[:,j], x_test[:,j] = normalize(x_train[:,j], x_test[:,j])
        pass

    # Append y test set to grand list
    y_test_all.append(y_test)

    # Fit Logistic Regression
    log_reg = LogisticRegression(penalty = 'l2', C = .01)
    log_reg.fit(x_train, y_train)

    # Create preditions probabilities append
    y_probs = log_reg.predict_proba(x_test)
    y_probs_all.append(y_probs[:,1])

    # Create predictions
    y_preds = log_reg.predict(x_test)
    acc = accuracy_score(y_test, y_preds)
    acc_all.append(acc)
    # ROC Curve
    fpr, tpr, thresholds = roc_curve(y_test, y_probs[:,1])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw = 2, alpha = 0.5,
             label='Fold {0} (AUC = {1:0.4f})'.format(i+1, roc_auc))

    i += 1
    pass

# Mean AUC
y_test_all = np.concatenate(y_test_all)
y_probs_all = np.concatenate(y_probs_all)
fpr, tpr, thresholds = roc_curve(y_test_all, y_probs_all)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, lw = 2,
             label='Mean ROC (AUC = {0:0.4f})'.format(roc_auc))

# Format ROC Plot
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
         label = 'Chance')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right", fontsize = 9)
plt.title('Logistic Regression: {}-Fold Cross Validation ROC Curve'.form
at(i))
plt.show()
```



In [25]:
```
# Average Accuracy
print('Average accuracy: {0:0.4}'.format(np.mean(acc_all)))
```

```
Average accuracy: 0.7874
```

In [26]:
```
# Confusion Matrix
pd.crosstab(y_test, y_preds, rownames=['True Class'], colnames=['Predict
ed Class'], margins=True)
```

Out[26]:

Predicted Class	**0**	**1**	**All**
True Class			
0	3107	444	3551
1	530	1482	2012
All	3637	1926	5563

(h) Describe how your system performed, where your supervised learning algorithm performed well, and where it did not, and how you could improve it.

Overall, our system preformed quite well with an AUC of .85 and an accuracy of .79. This performance is meaningfully better than chance, but of course there is still room to improve these measures.

Looking at our confusion matrix there is a false negative rate of ~15%. Thinking about the context of our problem, it is very costly for us to miss or ignore a segment people with potentially heightened propensity for suicide. In this case, it may make sense to adjust our decision threshold such that we have fewer false negatives in exchange for additional false positives.

(i) Write a brief summary / elevator pitch for this work that you would put on LinkedIn to describe this project to future employers. This should focus on the high level impact and importance and overall takeaways and not on the nitty-gritty details.

The World Health Organization (WHO) estimates that each year approximately one million people die from suicide. The first step in suicide prevention is suicide detection. This analysis examines global suicide rates and hopes to identify and predict above-average suicide rates among different cohorts globally, across the socio-economic spectrum.

This analysis attempts to predict above-average suicides rates across the globe from 1985-2016 using both country-specific predictors such as country, population, GDP as well as personal-level predictors such as age, generation, and sex. Using logistic regression and lasso regularization, we were able to achieve and AUC of .85 and an accuracy of .79. These results indicate our model can successfully predict above-average suicide rates significantly better than chance.