Homework 1: Applied Machine Learning Assignment

This assignment covers contents of the first three lectures.

We will be focusing on topics related to

1. Data Visualization and Analysis
2. Supervised Learning - Linear Regression, Logistic Regression, and SVM with Data Preprocessing.

Due Date is October 3, 11:59 PM.

Name: Rachel Peng (Xueheng Peng)

UNI: xp2197

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from numpy.linalg import inv
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import OrdinalEncoder
from sklearn.svm import LinearSVC, SVC
from sklearn.metrics import accuracy_score, mean_squared_error
```

In [2]:
```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

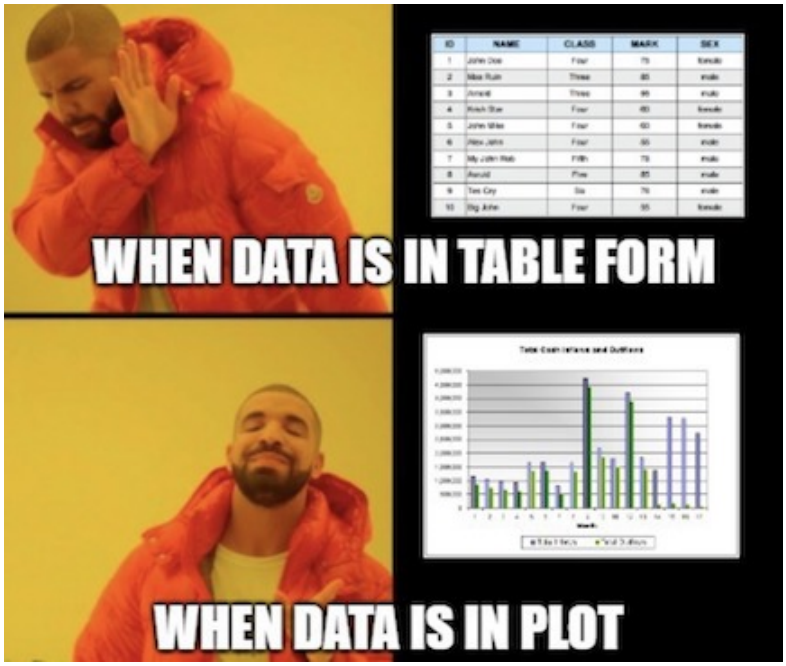
In [3]:
```
pd.options.mode.chained_assignment = None
```

Task 1: Data Visualization and Analysis

"Now that's A LOT of data. Can you show me something I can understand?"

This question often arises when we see datasets with thousands of rows and want to understand the characteristics of data.

Data visualization comes to our rescue!



We are going to use the credit-dataset for Task 1.

1.1 Plot the distribution of the features - credit_amount, age, and duration using a histogram. Make sure to label your axes while plotting. [6 points]

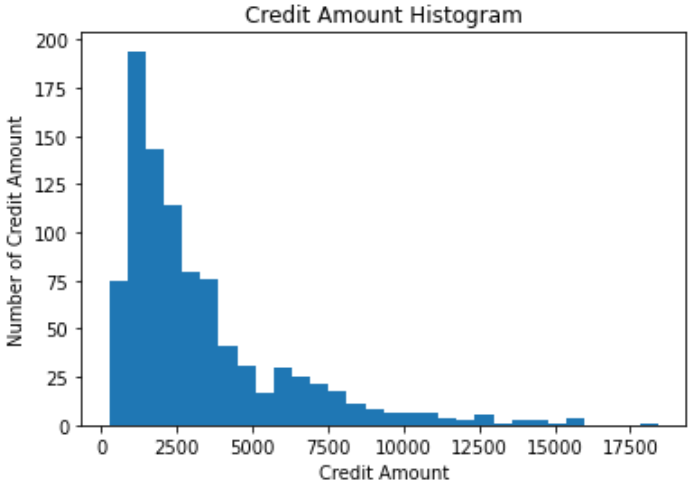
In [4]:
```
# YOUR CODE HERE
credit = pd.read_csv('dataset_credit.csv')
credit.head()
```

Out[4]:

	duration	credit_amount	savings_status	employment	property_magnitude	age	own_telephone	class
0	6	1169.0	'no known savings'	'>=7'	'real estate'	67	NaN	good
1	48	5951.0	'<100'	'1<=X<4'	'real estate'	22	NaN	bad
2	12	2096.0	'<100'	'4<=X<7'	'real estate'	49	NaN	good
3	42	7882.0	'<100'	'4<=X<7'	'life insurance'	45	none	good
4	24	4870.0	'<100'	'1<=X<4'	'no known property'	53	NaN	bad

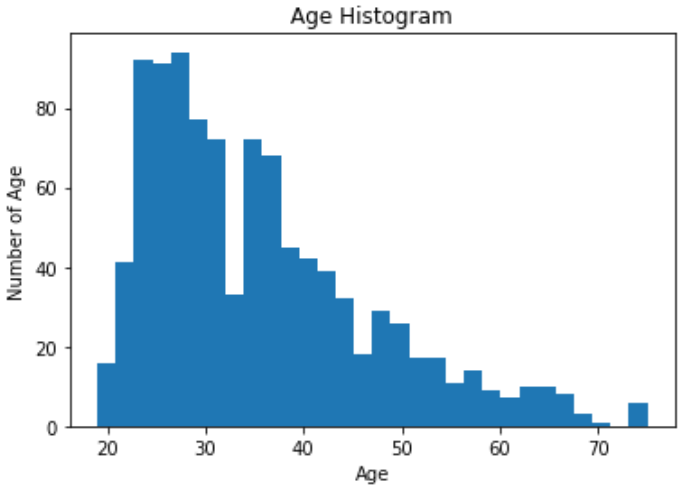
In [5]:
```
amount = credit['credit_amount']
amount.plot(kind='hist', bins=30, title='Credit Amount Histogram')
plt.xlabel('Credit Amount')
plt.ylabel('Number of Credit Amount')
```

Out[5]:
```
Text(0, 0.5, 'Number of Credit Amount')
```



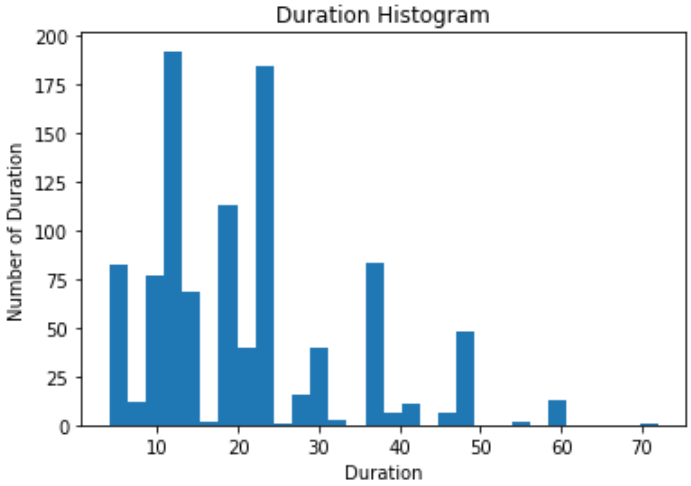
In [6]:
```
age = credit['age']
age.plot(kind='hist', bins=30, title='Age Histogram')
plt.xlabel('Age')
plt.ylabel('Number of Age')
```

Out[6]:
```
Text(0, 0.5, 'Number of Age')
```



In [7]:
```
duration = credit['duration']
duration.plot(kind='hist', bins=30, title='Duration Histogram')
plt.xlabel('Duration')
plt.ylabel('Number of Duration')
```

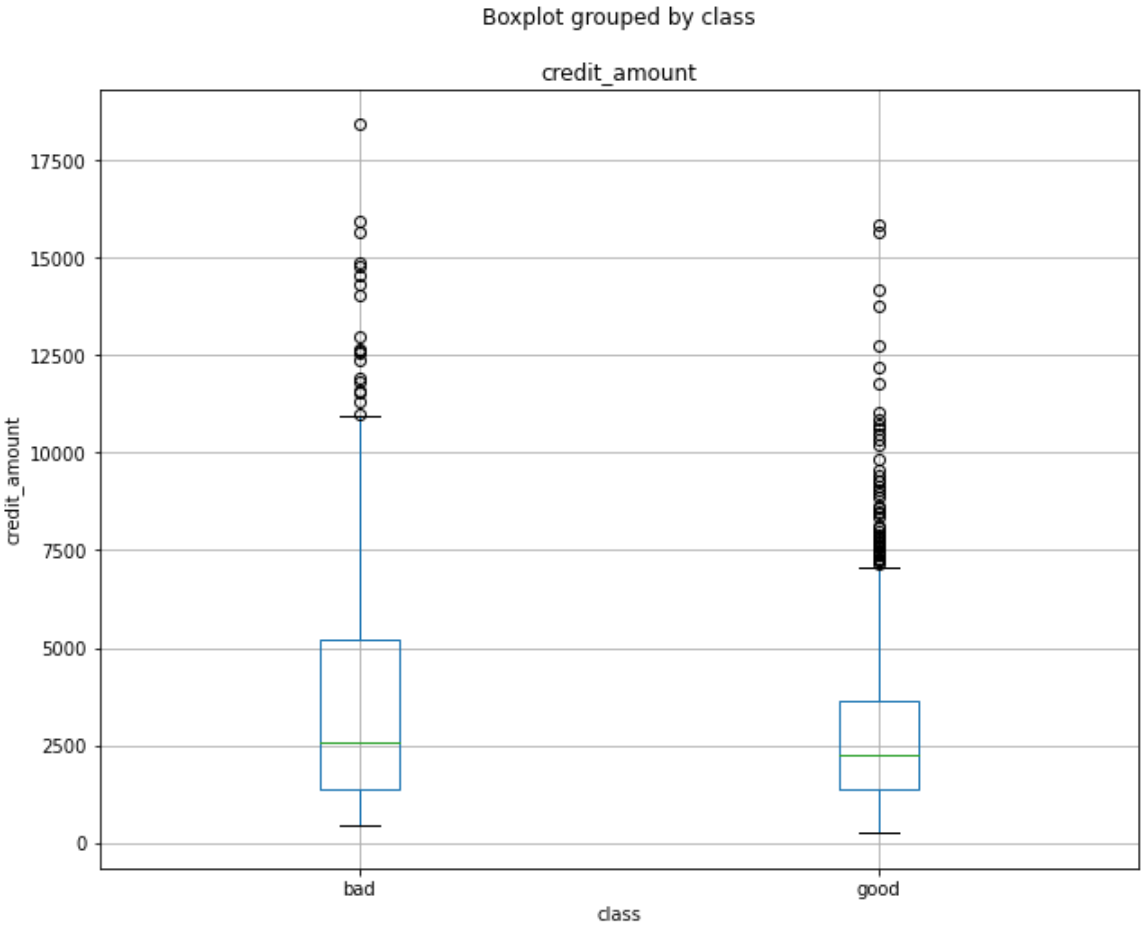
Out[7]:
```
Text(0, 0.5, 'Number of Duration')
```



1.2 Plot the relationships between the features - class and credit_amount using box plots. Make sure to label the axes[4 points]

In [8]:
```
# Your code here
credit_class = credit[['credit_amount', 'class']]
plot = credit_class.boxplot(column="credit_amount", by="class", figsize=(10, 8))
plot.set_ylabel('credit_amount')
```

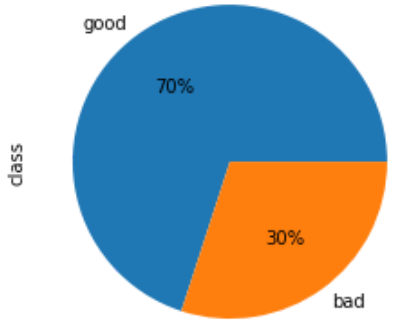
Out[8]:
```
Text(0, 0.5, 'credit_amount')
```



1.3 Plot the distribution of label 'class' using a pie chart. Be sure to label correctly. What do you infer about the data and its distribution from all the plots? (1.1, 1.2, and 1.3)[5 points]

In [102]:
```
# Your code here
class_data = credit['class'].value_counts()
class_data.plot(kind='pie', autopct='%.0f%%')
```

Out[102]:
```
<AxesSubplot:ylabel='class'>
```

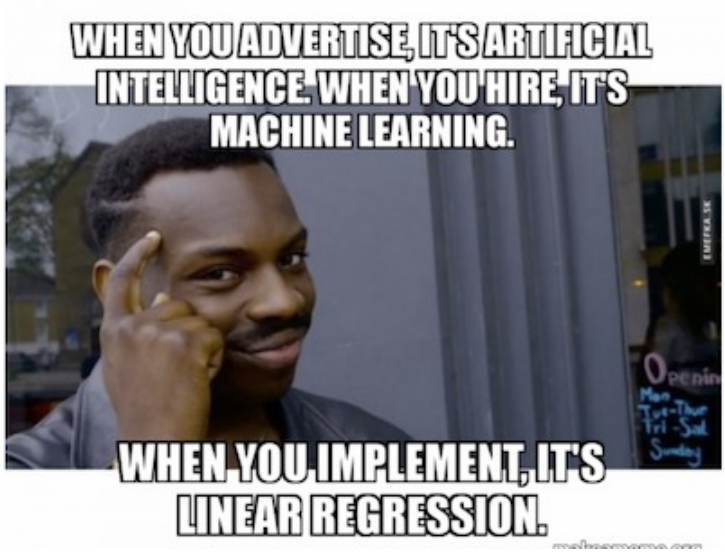


The pie chart above shows that the data distribution is not even for this dataset: class good has more than doubled the data than class bad.
Histograms from 1.1 suggest that credit amount, age, and duration data are skewed to the right, meaning that they don't fall into normal distribution, and their mode is less than mean.
Boxplots from 1.2 suggest that both the medians of credit amount for good and bad class are close, but bad class has larger variance than good class, for it has a wider range from the median to Q3 and to max/outliers.

Task 2 : Linear Models for Regression and Classification

In this notebook, we will be implementing three linear models **linear regression, logistic regression, and SVM**. We will see that despite some of their differences at the surface, these linear models (and many machine learning models in general) are fundamentally doing the same thing - that is, optimizing model parameters to minimize a loss function on data.

Part 1: Linear Regression



In part 1, we will use two datasets - synthetic and auto-mpg to train and evaluate our linear regression model.

The first dataset will be a synthetic dataset sampled from the following equations:

$\epsilon \sim Normal(0, 3)$

$y = 5x + 10 + \epsilon$

In [10]:
```
np.random.seed(0)
epsilon = np.random.normal(0, 3, 100)
x = np.linspace(0, 10, 100)
# y = np.linspace(0, 5, 100)
y = 5 * x + 10 + epsilon
```

To apply linear regression, we need to first check if the assumptions of linear regression are not violated.

Assumptions of Linear Regression:

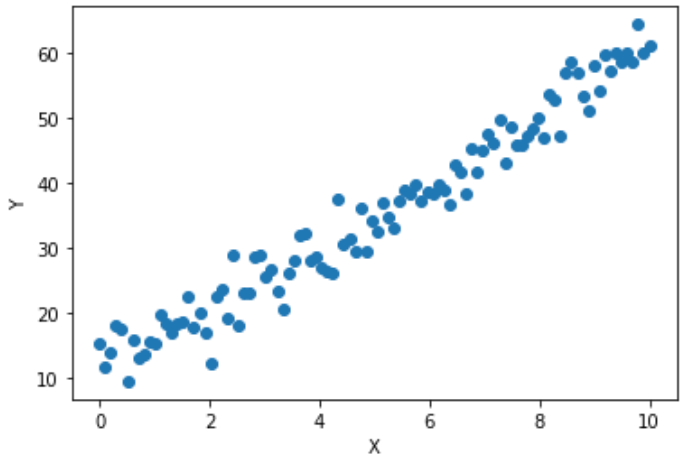
- Linearity: is a linear (technically affine) function of x.
- Independence: the x's are independently drawn, and not dependent on each other.
- Homoscedasticity: the ϵ's, and thus the y's, have constant variance.
- Normality: the ϵ's are drawn from a Normal distribution (i.e. Normally-distributed errors)

These properties, as well as the simplicity of this dataset, will make it a good test case to check if our linear regression model is working properly.

2.1.1 Plot y vs x in the synthetic dataset as a scatter plot. Label your axes and make sure your y-axis starts from 0. Do the features have linear relationship?[2 points]

In [11]:
```
# Your code here
plt.scatter(x, y)
plt.xlabel('X')
plt.ylabel('Y')
```

Out[11]:
```
Text(0, 0.5, 'Y')
```



The figure above and how x and y are defined show that they do have a linear relationship. Linearity is satisfied through y = 5 * x + 10 + epsilon;

Independence is satisfied because x = np.linspace(0, 10, 100), meaning that x is 100 values evenly spaced over the range of 0 to 10, so those values can't be dependent;

Homoscedasticity is satisfied because epsilon is drawn from a normal distribution: epsilon = np.random.normal(0, 3, 100), and the standard deviation is set at a constant of 3, so epsilon has constant variance;

Normality is satisfided because epsilon is drawn from a normal distribution.

The second dataset we will be using is an auto MPG dataset (https://archive.ics.uci.edu/ml/datasets/Auto+MPG). This dataset contains various characteristics for around 400 cars. We will use linear regression to predict the mpg label from seven features (4 continuous, 3 discrete).

In [12]:
```
# Load auto MPG dataset
auto_mpg_df = pd.read_csv('auto-mpg.csv')

# drop some rows with missing entries
auto_mpg_df = auto_mpg_df[auto_mpg_df['horsepower'] != '?']

# Cast horsepower column to float
auto_mpg_df['horsepower'] = auto_mpg_df['horsepower'].astype(float)

auto_mpg_df
```

Out[12]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin
0	18.0	8	307.0	130.0	3504.0	12.0	70	1
1	15.0	8	350.0	165.0	3693.0	11.5	70	1
2	18.0	8	318.0	150.0	3436.0	11.0	70	1
3	16.0	8	304.0	150.0	3433.0	12.0	70	1
4	17.0	8	302.0	140.0	3449.0	10.5	70	1
...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86.0	2790.0	15.6	82	1
394	44.0	4	97.0	52.0	2130.0	24.6	82	2
395	32.0	4	135.0	84.0	2295.0	11.6	82	1
396	28.0	4	120.0	79.0	2625.0	18.6	82	1
397	31.0	4	119.0	82.0	2720.0	19.4	82	1

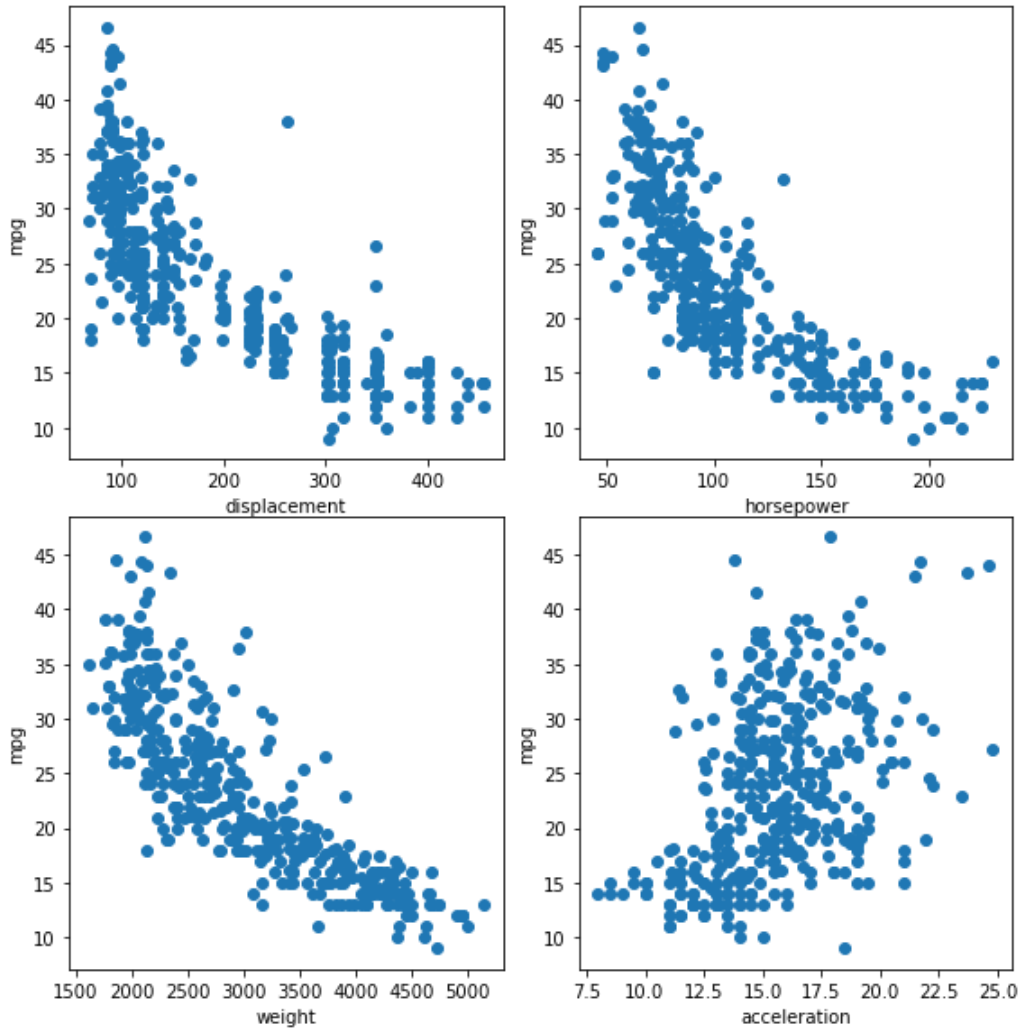
392 rows × 8 columns

In [13]:
```
# Split data into features and labels
auto_mpg_X = auto_mpg_df.drop(columns=['mpg'])
auto_mpg_y = auto_mpg_df['mpg']
```

2.1.2 Plot the relationships between the label (mpg) and the continuous features (displacement, horsepower, weight, acceleration) using a small multiple of scatter plots. Make sure to label the axes.[4 points]

In [14]:
```
# Your code here
fig, axs = plt.subplots(2, 2, figsize=(8, 8))
fig.tight_layout()
axs[0, 0].scatter(auto_mpg_X['displacement'], auto_mpg_y)
axs[0, 0].set(xlabel='displacement', ylabel='mpg')
axs[0, 1].scatter(auto_mpg_X['horsepower'], auto_mpg_y)
axs[0, 1].set(xlabel='horsepower', ylabel='mpg')
axs[1, 0].scatter(auto_mpg_X['weight'], auto_mpg_y)
axs[1, 0].set(xlabel='weight', ylabel='mpg')
axs[1, 1].scatter(auto_mpg_X['acceleration'], auto_mpg_y)
axs[1, 1].set(xlabel='acceleration', ylabel='mpg')
```

Out[14]:
```
[Text(0.5, 51.00000000000006, 'acceleration'),
 Text(288.49090909090904, 0.5, 'mpg')]
```

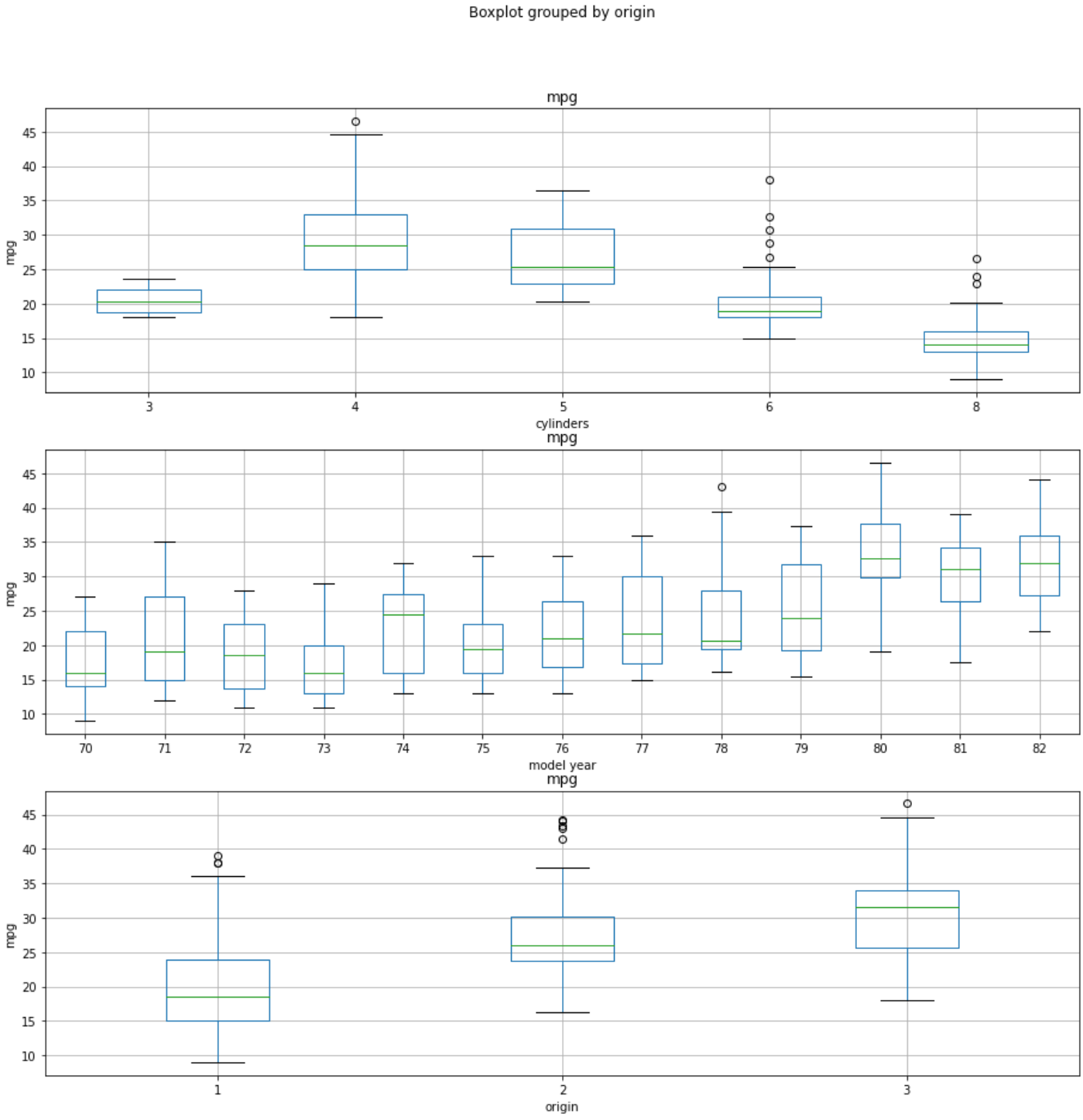


2.1.3 Plot the relationships between the label (mpg) and the discrete features (cylinders, model year, origin) using a small multiple of box plots. Make sure to label the axes.[3 points]

In [15]:
```
# Your code here
fig, axs = plt.subplots(3, 1, figsize=(15, 15))

auto_mpg_df.boxplot(column='mpg', by='cylinders', ax=axs[0])
axs[0].set_ylabel('mpg')
auto_mpg_df.boxplot(column='mpg', by='model year', ax=axs[1])
axs[1].set_ylabel('mpg')
auto_mpg_df.boxplot(column='mpg', by='origin', ax=axs[2])
axs[2].set_ylabel('mpg')
```

Out[15]:
```
Text(0, 0.5, 'mpg')
```



2.1.4 From the visualizations above, do you think linear regression is a good model for this problem? Why and/or why not? Please explain.[2 points]

Your answer here

I think linear regression is a good model for this problem, as the plots above all show general linearly increasing or decreasing trends, although some maybe have stronger linear correlation than others.

Data Preprocessing

Before we can fit a linear regression model, there are several pre-processing steps we should apply to the datasets:

1. Encode categorial features appropriately.
2. Split the dataset into training (60%), validation (20%), and test (20%) sets.
3. Standardize the columns in the feature matrices X_train, X_val, and X_test to have zero mean and unit variance. To avoid information leakage, learn the standardization parameters (mean, variance) from X_train, and apply it to X_train, X_val, and X_test.
4. Add a column of ones to the feature matrices X_train, X_val, and X_test. This is a common trick so that we can learn a coefficient for the bias term of a linear model.

The processing steps on the synthetic dataset have been provided for you below as a reference:

In [23]:
```
X = x.reshape((100, 1))   # Turn the x vector into a feature matrix X

# 1. No categorical features in the synthetic dataset (skip this step)

# 2. Split the dataset into training (60%), validation (20%), and test (20%) sets
X_dev, X_test, y_dev, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X_dev, y_dev, test_size=0.25, random_state=0)

# 3. Standardize the columns in the feature matrices
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)   # Fit and transform scalar on X_train
X_val = scaler.transform(X_val)           # Transform X_val
X_test = scaler.transform(X_test)         # Transform X_test

# 4. Add a column of ones to the feature matrices
X_train = np.hstack([np.ones((X_train.shape[0], 1)), X_train])
X_val = np.hstack([np.ones((X_val.shape[0], 1)), X_val])
X_test = np.hstack([np.ones((X_test.shape[0], 1)), X_test])

print(X_train[:5], '\n\n', y_train[:5])
```

```
[[ 1.          0.53651502]
 [ 1.         -1.00836082]
 [ 1.         -0.72094206]
 [ 1.         -0.25388657]
 [ 1.          0.64429705]]

 [38.44273829 19.38966655 26.79105322 30.69326568 45.00432104]
```

2.1.5 Apply the same processing steps on the auto MPG dataset.[3 points]

In [24]:
```
# Your code here

# Split the dataset into training (60%), validation (20%), and test (20%) sets
X_dev, auto_mpg_X_test, y_dev, auto_mpg_y_test = train_test_split(auto_mpg_X, auto_mpg_y, test_
auto_mpg_X_train, auto_mpg_X_val, auto_mpg_y_train, auto_mpg_y_val = train_test_split(X_dev, y_

# Standardize the columns in the feature matrices
scaler = StandardScaler()
auto_mpg_X_train = scaler.fit_transform(auto_mpg_X_train)   # Fit and transform scalar on auto_
auto_mpg_X_val = scaler.transform(auto_mpg_X_val)           # Transform auto_mpg_X_val
auto_mpg_X_test = scaler.transform(auto_mpg_X_test)         # Transform auto_mpg_X_test

# Add a column of ones to the feature matrices
auto_mpg_X_train = np.hstack([np.ones((auto_mpg_X_train.shape[0], 1)), auto_mpg_X_train])
auto_mpg_X_val = np.hstack([np.ones((auto_mpg_X_val.shape[0], 1)), auto_mpg_X_val])
auto_mpg_X_test = np.hstack([np.ones((auto_mpg_X_test.shape[0], 1)), auto_mpg_X_test])

print(auto_mpg_X_train[:5], '\n\n', auto_mpg_y_train[:5])
```

```
[[ 1.          0.37998163  0.39492947  0.1100916   0.8241919   0.28262047
  -0.57603817 -0.77559006]
 [ 1.         -0.83804168 -0.97348359 -0.87531843 -1.20346504 -0.54674887
  -0.02809942  0.43433043]
 [ 1.          1.59800495  1.33761402  1.37704734  1.02260224 -1.159761
  -0.85000755 -0.77559006]
 [ 1.         -0.83804168 -0.5173459  -0.48115442 -0.53443504 -0.00585582
   1.34174745 -0.77559006]
 [ 1.         -0.83804168 -0.97348359 -1.49471902 -1.0244118   2.15771638
   1.06777808  0.43433043]]

 135    18.0
197    29.0
89     15.0
338    27.2
325    44.3
Name: mpg, dtype: float64
```

At the end of this pre-processing, you should have the following vectors and matrices:

- Syntheic dataset: X_train, X_val, X_test, y_train, y_val, y_test
- Auto MPG dataset: auto_mpg_X_train, auto_mpg_X_val, auto_mpg_X_test, auto_mpg_y_train, auto_mpg_y_val, auto_mpg_y_test

Implement Linear Regression

Now, we can implement our linear regression model! Specifically, we will be implementing ridge regression, which is linear regression with L2 regularization. Given an (m x n) feature matrix X, an (m x 1) label vector y, and an (n x 1) weight vector w, the hypothesis function for linear regression is:

$$y = Xw$$

Note that we can omit the bias term here because we have included a column of ones in our X matrix, so the bias term is learned implicitly as a part of w. This will make our implementation easier.

Our objective in linear regression is to learn the weights w which best fit the data. This notion can be formalized as finding the optimal w which minimizes the following loss function:

$$\min_{w} \|Xw - y\|_2^2 + \alpha\|w\|_2^2$$

This is the ridge regression loss function. The $\|Xw - y\|_2^2$ term penalizes predictions Xw which are not close to the label y. And the $\alpha\|w\|_2^2$ penalizes large weight values, to favor a simpler, more generalizable model. The α hyperparameter, known as the regularization parameter, is used to tune the complexity of the model - a higher α results in smaller weights and lower complexity, and vice versa. Setting $\alpha = 0$ gives us vanilla linear regression.

Conveniently, ridge regression has a closed-form solution which gives us the optimal w without having to do iterative methods such as gradient descent. The closed-form solution, known as the Normal Equations, is given by:

$$w = (X^T X + \alpha I)^{-1} X^T y$$

2.1.6 Implement a `LinearRegression` class with two methods: `train` and `predict` .[8 points] You may NOT use sklearn for this implementation. You may, however, use `np.linalg.solve` to find the closed-form solution. It is highly recommended that you vectorize your code.

In [25]:
```
class LinearRegression():
    '''
    Linear regression model with L2-regularization (i.e. ridge regression).

    Attributes
    ----------
    alpha: regularization parameter
    w: (n x 1) weight vector
    '''

    def __init__(self, alpha=0):
        self.alpha = alpha
        self.w = None

    def train(self, X, y):
        '''Trains model using ridge regression closed-form solution
        (sets w to its optimal value).

        Parameters
        ----------
        X : (m x n) feature matrix
        y: (m x 1) label vector

        Returns
        -------
        None
        '''
        ### Your code here
        dimension = X.shape[1]
        identity = np.identity(dimension)
        self.w = np.linalg.inv(X.T.dot(X) + self.alpha*identity).dot(X.T).dot(y)


    def predict(self, X):
        '''Predicts on X using trained model.

        Parameters
        ----------
        X : (m x n) feature matrix

        Returns
        -------
        y_pred: (m x 1) prediction vector
        '''
        y_pred = X.dot(self.w)
        return y_pred
```

Train, Evaluate, and Interpret Linear Regression Model

2.1.7 Using your `LinearRegression` implementation above, train a vanilla linear regression model ($\alpha = 0$) on (X_train, y_train) from the synthetic dataset. Use this trained model to predict on X_test. Report the first 3 and last 3 predictions on X_test, along with the actual labels in y_test.[3 points]

In [27]:
```
# Your code here
ridge = LinearRegression()
ridge.train(X_train, y_train)
prediction = ridge.predict(X_test)
prediction
```

Out[27]:
```
array([23.29684501, 53.01355017, 11.41016295, 37.65991917, 47.56548756,
       56.4804991 , 18.34406082, 46.57493072, 37.16464075, 57.47105594,
       36.66936233, 55.98522068, 49.05132281, 16.85822556, 13.88655504,
       25.27795869, 21.31573133, 22.30628817, 26.76379395, 14.38183346])
```

In [28]:
```
y_test[0:3]
```

Out[28]:
```
array([23.26858868, 56.97068215, 13.94631496])
```

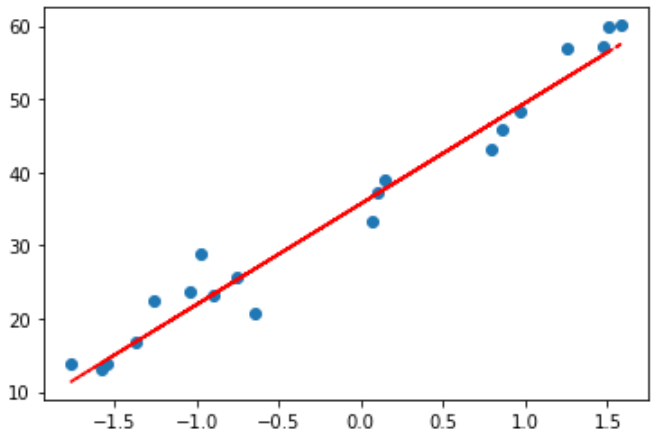
In [29]:
```
y_test[-3:]
```

Out[29]:
```
array([28.93047599, 20.72427726, 13.73074749])
```

First 3 predictions: 23.29684501, 53.01355017, 11.41016295
Last 3 predictions: 22.30628817, 26.76379395, 14.38183346
Actual labels in y_test:
First 3: 23.26858868, 56.97068215, 13.94631496
Last 3: 28.93047599, 20.72427726, 13.73074749

2.1.8 Plot a scatter plot of y_test vs X_test (just the non-ones column). Then, using the weights from the trained model above, plot the best-fit line for this data on the same figure.[2 points] If your line goes through the data points, you have likely implemented the linear regression correctly!

In [30]:
```
# Your code here
plt.figure()
plt.scatter(X_test[:, 1], y_test)
plt.plot(X_test[:, 1:], prediction, '--', color='r')
plt.show()
```



2.1.9 Train a linear regression model ($\alpha = 0$) on the auto MPG training data. Make predictions and report the mean-squared error (MSE) on the training, validation, and test sets. Report the first 3 and last 3 predictions on the test set, along with the actual labels.[4 points]

In [32]:
```
# Your code here
ridge_mpg = LinearRegression()
ridge_mpg.train(auto_mpg_X_train, auto_mpg_y_train)
prediction_xtrain = ridge_mpg.predict(auto_mpg_X_train)
prediction_xval = ridge_mpg.predict(auto_mpg_X_val)
prediction_xtest = ridge_mpg.predict(auto_mpg_X_test)
mse_train = mean_squared_error(auto_mpg_y_train, prediction_xtrain)
mse_val = mean_squared_error(auto_mpg_y_val, prediction_xval)
mse_test = mean_squared_error(auto_mpg_y_test, prediction_xtest)
print(mse_train, mse_val, mse_test)
```

```
10.67058419333088 12.944798748782656 10.881879498129635
```

The MSEs for train, validation, and test are 10.67058419333088, 12.944798748782656, 10.881879498129635.

In [33]:
```
prediction_xtest[0:3]
```

Out[33]:
```
array([26.3546854 , 25.49133646, 10.15877236])
```

In [34]:
```
prediction_xtest[-3:]
```

Out[34]:
```
array([26.85946741, 21.85952894, 32.03222623])
```

In [35]:
```
auto_mpg_y_test[0:3]
```

Out[35]:
```
146    28.0
282    22.3
69     12.0
Name: mpg, dtype: float64
```

In [36]:
```
auto_mpg_y_test[-3:]
```

Out[36]:
```
56     26.0
262    19.2
216    31.5
Name: mpg, dtype: float64
```

First 3 predictions: 26.3546854 , 25.49133646, 10.15877236
Last 3 predictions: 26.85946741, 21.85952894, 32.03222623
Actual labels in auto_mpg_y_test:
First 3: 28.0, 22.3, 12.0
Last 3: 26.0, 19.2, 31.5

2.1.10 As a baseline model, use the mean of the training labels (auto_mpg_y_train) as the prediction for all instances. Report the mean-squared error (MSE) on the training, validation, and test sets using this baseline. [3 points] This is a common baseline used in regression problems and tells you if your model is any good. Your linear regression MSEs should be much lower than these baseline MSEs.

In [37]:
```
# Your code here
mean = sum(auto_mpg_y_train)/len(auto_mpg_y_train)
mean_train = [mean]*len(auto_mpg_y_train)
mean_val = [mean]*len(auto_mpg_y_val)
mean_test = [mean]*len(auto_mpg_y_test)
mse_mean_train = mean_squared_error(auto_mpg_y_train, mean_train)
mse_mean_val = mean_squared_error(auto_mpg_y_val, mean_val)
mse_mean_test = mean_squared_error(auto_mpg_y_test, mean_test)
print(mse_mean_train, mse_mean_val, mse_mean_test)
```

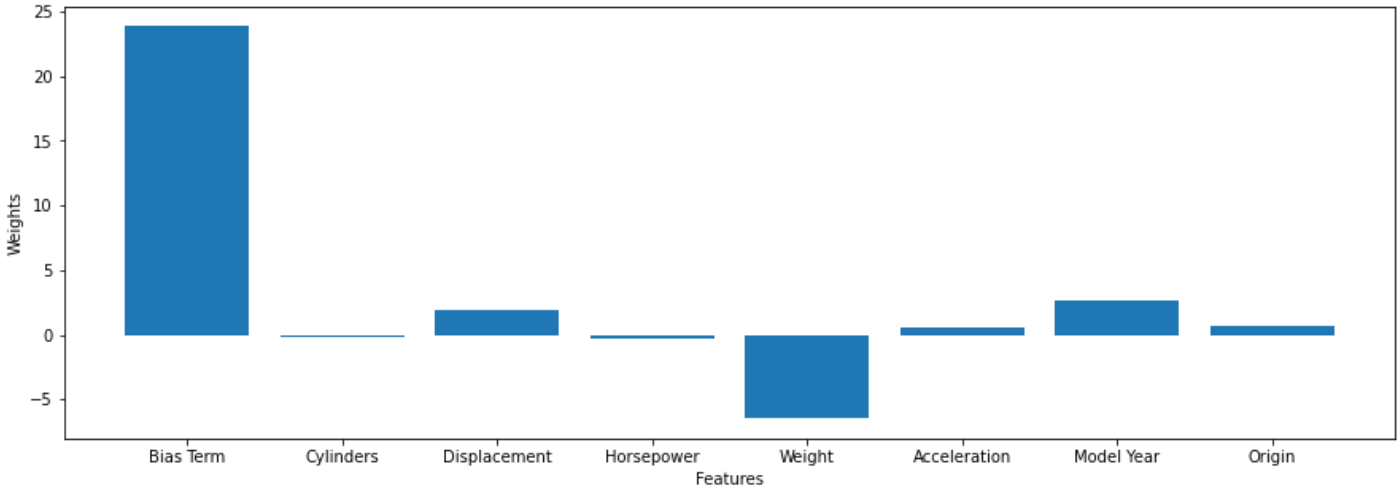
```
60.56461465410184 60.47988929483246 62.46160518794076
```

The MSEs for train, validation, and test are 60.56461465410184, 60.47988929483246, 62.46160518794076, which are all much higher than the linear regression MSEs.

2.1.11 Interpret your model trained on the auto MPG dataset using a bar chart of the model weights. [3 points] Make sure to label the bars (x-axis) and don't forget the bias term!

In [38]:
```
# Your code here
w_names = ['Bias Term', 'Cylinders', 'Displacement', 'Horsepower', 'Weight', 'Acceleration', 'M
plt.figure(figsize=(15, 5))
plt.bar(w_names, ridge_mpg.w)
plt.xlabel('Features')
plt.ylabel('Weights')
```

Out[38]:
```
Text(0, 0.5, 'Weights')
```



2.1.12 According to your model, which features are the greatest contributors to the MPG?[2 points]

The car weight, model year, and displacement are the greatest contributors to the MPG, judging from the height of the bars.

Tune Regularization Parameter α

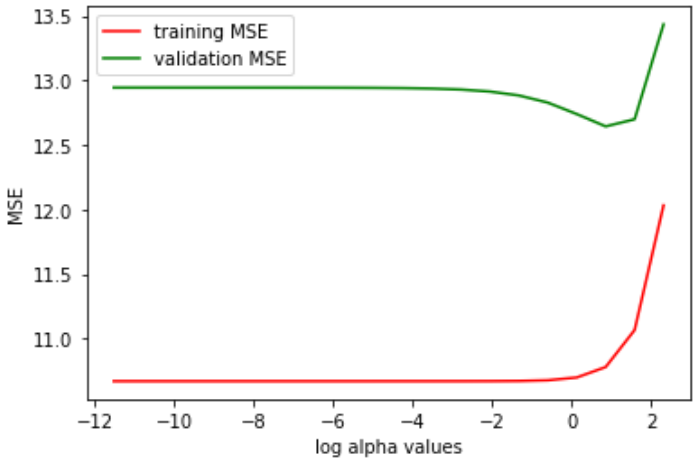
Now, let's do ridge regression and tune the α regularization parameter on the auto MPG dataset.

2.1.13 Sweep out values for α using alphas = np.logspace(-5, 1, 20). Perform a grid search over these α values, recording the training and validation MSEs for each α. A simple grid search is fine, no need for k-fold cross validation. Plot the training and validation MSEs as a function of α on a single figure. Make sure to label the axes and the training and validation MSE curves. Use a log scale for the x-axis.[4 points]

In [39]:
```
# Your code here
alphas = np.logspace(-5, 1, 20)
mses_train = []
mses_val = []
for a in alphas:
    ridge_tune = LinearRegression(a)
    ridge_tune.train(auto_mpg_X_train, auto_mpg_y_train)
    prediction_xtrain = ridge_tune.predict(auto_mpg_X_train)
    prediction_xval = ridge_tune.predict(auto_mpg_X_val)
    mse_train = mean_squared_error(auto_mpg_y_train, prediction_xtrain)
    mses_train.append(mse_train)
    mse_val = mean_squared_error(auto_mpg_y_val, prediction_xval)
    mses_val.append(mse_val)
print(alphas)
alphas = np.log(alphas)
```

```
[1.00000000e-05 2.06913808e-05 4.28133240e-05 8.85866790e-05
 1.83298071e-04 3.79269019e-04 7.84759970e-04 1.62377674e-03
 3.35981829e-03 6.95192796e-03 1.43844989e-02 2.97635144e-02
 6.15848211e-02 1.27427499e-01 2.63665090e-01 5.45559478e-01
 1.12883789e+00 2.33572147e+00 4.83293024e+00 1.00000000e+01]
```

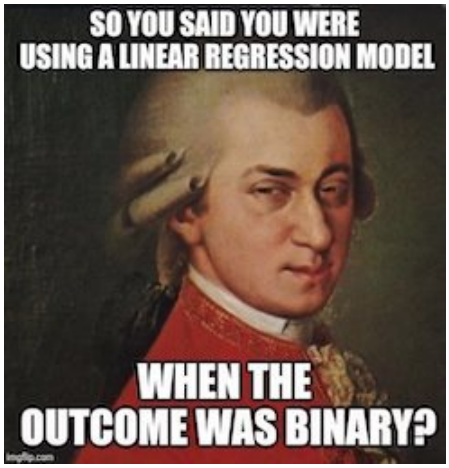
In [40]:
```
plt.figure()
plt.plot(alphas, mses_train, color='r', label='training MSE')
plt.plot(alphas, mses_val, color='g', label='validation MSE')
plt.xlabel('log alpha values')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



2.1.14 Explain your plot above. How do training and validation MSE behave with decreasing model complexity (increasing α)?[2 points]

Training MSE remains constant until it increases when the model uses alpha log values higher than 0.
Validation MSE remains constant, but it decreases to its min value earlier than training MSE starts to increase. Then it increases after alpha log value of 0.5, roughly.
A higher alpha value means the model penalizes the optimization function more and larger smoothness constraint. Since the model's flexibility of the fit becomes stricter with higher alpha values, the training MSE increases.
Idealy, we would use the alpha value that gives the leaset validaion MSE to refit our ridge model.

Part 2: Logistic Regression



In this part we would use Logistic Regression on NBA rookie stats to predict if player will last 5 years in league

Class variable represent: y = 0 if career years played < 5 y = 1 if career years played >= 5

	Description
Name	Name
GP	Games Played
MIN	MinutesPlayed
PTS	PointsPerGame
FGM	FieldGoalsMade
FGA	FieldGoalAttempts
FG%	FieldGoalPercent
3P Made	3PointMade
3PA	3PointAttempts
3P%	3PointAttempts
FTM	FreeThrowMade
FTA	FreeThrowAttempts
FT%	FreeThrowPercent
OREB	OffensiveRebounds
DREB	DefensiveRebounds
REB	Rebounds
AST	Assists
STL	Steals
BLK	Blocks
TOV	Turnovers
TARGET_5Yrs	Outcome: 1 if career length >= 5 yrs, 0 if < 5...

In [41]:
```
nba_reg = pd.read_csv("nba_logreg.csv")
nba_reg.head()
```

Out[41]:

	Name	GP	MIN	PTS	FGM	FGA	FG%	3P Made	3PA	3P%	...	FTA	FT%	OREB	DREB	REB	AST	STL	BLK	TOV
0	Brandon Ingram	36	27.4	7.4	2.6	7.6	34.7	0.5	2.1	25.0	...	2.3	69.9	0.7	3.4	4.1	1.9	0.4	0.4	1.3
1	Andrew Harrison	35	26.9	7.2	2.0	6.7	29.6	0.7	2.8	23.5	...	3.4	76.5	0.5	2.0	2.4	3.7	1.1	0.5	1.6
2	JaKarr Sampson	74	15.3	5.2	2.0	4.7	42.2	0.4	1.7	24.4	...	1.3	67.0	0.5	1.7	2.2	1.0	0.5	0.3	1.0
3	Malik Sealy	58	11.6	5.7	2.3	5.5	42.6	0.1	0.5	22.6	...	1.3	68.9	1.0	0.9	1.9	0.8	0.6	0.1	1.0
4	Matt Geiger	48	11.5	4.5	1.6	3.0	52.4	0.0	0.1	0.0	...	1.9	67.4	1.0	1.5	2.5	0.3	0.3	0.4	0.8

5 rows × 21 columns

In [42]:
```
nba_reg.shape
```

Out[42]:
```
(1340, 21)
```

Missing Value analysis

2.2.1 Are there any missing values in the dataset? If so, what can be done about it? (Think if removing is an option?) (Note: Name your dataset as nba_reg_new after removing NAs) [2 points]

There are missing values in the dataset because the following check returns true, and all missing values are in column 3P%.

In [43]:
```
# Your code here
# Check if there are missing values in the dataset
nan = nba_reg.isnull().values.any()
print(nan)
count = nba_reg.isnull().sum()
print(count)
```

```
True
Name            0
GP              0
MIN             0
PTS             0
FGM             0
FGA             0
FG%             0
3P Made         0
3PA             0
3P%            11
FTM             0
FTA             0
FT%             0
OREB            0
DREB            0
REB             0
AST             0
STL             0
BLK             0
TOV             0
TARGET_5Yrs     0
dtype: int64
```

In [44]:
```
nba_reg.shape
```

Out[44]:
```
(1340, 21)
```

In [45]:
```
nan_rows = nba_reg[nba_reg.isnull().any(axis=1)]
nan_rows[['3P Made', '3PA', '3P%']]
```

Out[45]:

	3P Made	3PA	3P%
338	0.0	0.0	NaN
339	0.0	0.0	NaN
340	0.0	0.0	NaN
358	0.0	0.0	NaN
386	0.0	0.0	NaN
397	0.0	0.0	NaN
507	0.0	0.0	NaN
509	0.0	0.0	NaN
510	0.0	0.0	NaN
521	0.0	0.0	NaN
559	0.0	0.0	NaN

There're 11 rows with NaN values in a dataset that has 1,000+ rows, but 3P% can be calculated using (3P Made/3PA)*100%. Those 3P% values are NaN becuase those players didn't attempt any 3-point shots, giving the formula a zero denominator. Changing those NaN values to 0.0 would be good and resonable.

In [46]:
```
# Your Code here
nba_reg_new = nba_reg.fillna(0.0)
# check
print(nba_reg_new.isnull().values.any())
print(nba_reg_new.shape)
```

```
False
(1340, 21)
```

2.2.2 Do you think that the distribution of labels is balanced? Why/why not? Hint: Find the probability of the different categories.[3 points]

In [47]:
```
# Your code here
targeted = sum(nba_reg_new['TARGET_5Yrs'] == 1.0)
print(targeted, nba_reg_new.shape[0]-targeted)
print(targeted/nba_reg_new.shape[0], 1-targeted/nba_reg_new.shape[0])
```

```
831 509
0.6201492537313433 0.37985074626865667
```

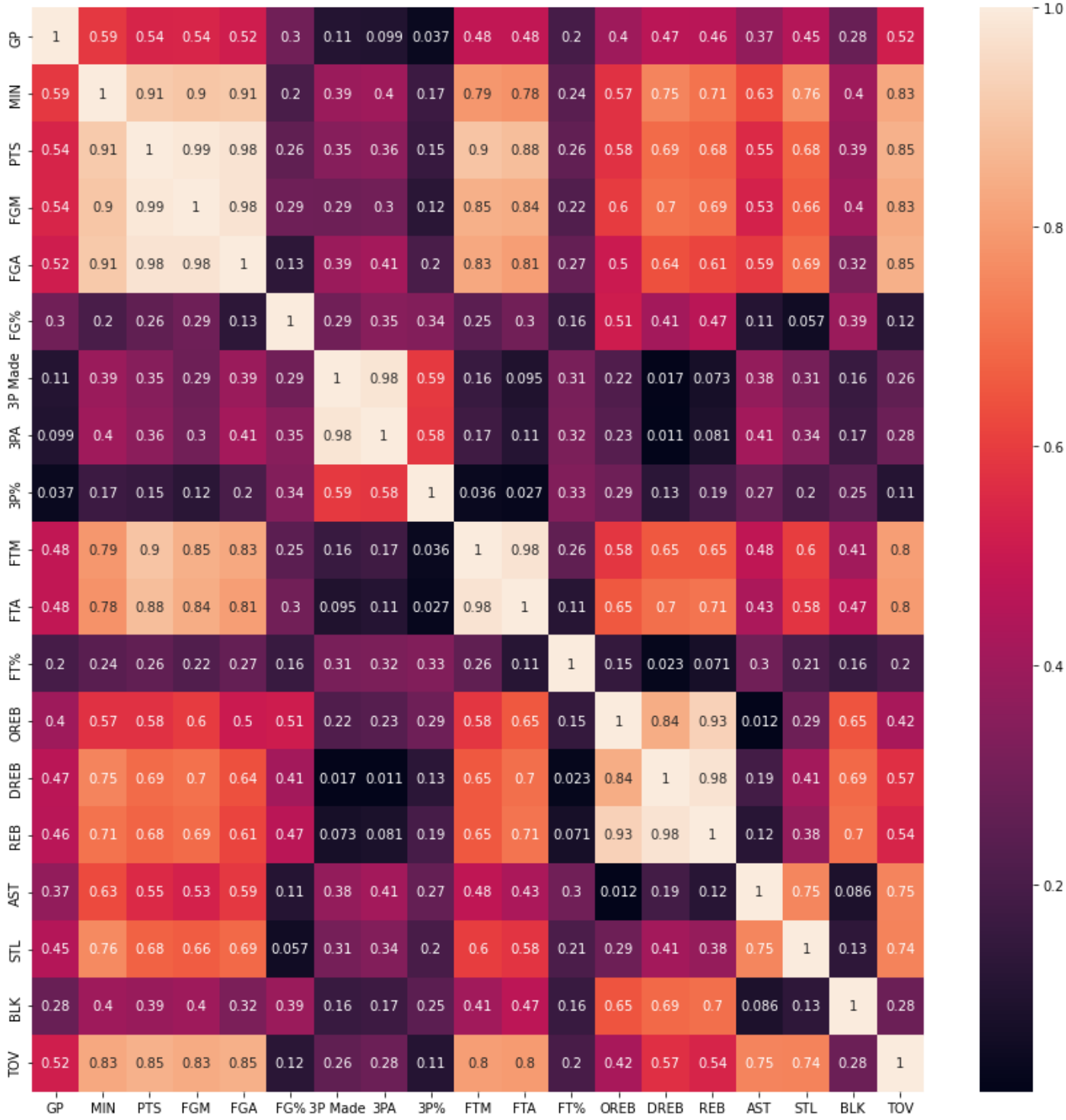
The distribution of labels is not balanced, as the computation above has shown that targeted players make 62% of the dataset, while 38% are untargeted players. There're less data to train on untargeted players.

In [48]:
```
nba_X = nba_reg_new.drop(columns=['TARGET_5Yrs'])
nba_y = nba_reg_new['TARGET_5Yrs']
print(nba_X.shape)
```

```
(1340, 20)
```

2.2.3 Plot the correlation matrix, and check if there is high correlation between the given numerical features (Threshold >=0.9). If yes, drop those highly correlated features from the dataframe. Why is necessary to drop those columns before proceeding further?[4 points]

In [49]:
```
# Your code here
corr = nba_X.corr().abs()
plt.figure(figsize=(15, 15))
sns.heatmap(corr, annot=True)
plt.show()
```



In [50]:
```
upper = corr.where(np.triu(np.ones(corr.shape), k=1).astype('bool'))
drop = [column for column in upper.columns if any(upper[column] > 0.9)]
nba_X.drop(drop, axis=1, inplace=True)
nba_X.head()
```

Out[50]:

	Name	GP	MIN	FG%	3P Made	3P%	FTM	FT%	OREB	DREB	AST	STL	BLK	TOV
0	Brandon Ingram	36	27.4	34.7	0.5	25.0	1.6	69.9	0.7	3.4	1.9	0.4	0.4	1.3
1	Andrew Harrison	35	26.9	29.6	0.7	23.5	2.6	76.5	0.5	2.0	3.7	1.1	0.5	1.6
2	JaKarr Sampson	74	15.3	42.2	0.4	24.4	0.9	67.0	0.5	1.7	1.0	0.5	0.3	1.0
3	Malik Sealy	58	11.6	42.6	0.1	22.6	0.9	68.9	1.0	0.9	0.8	0.6	0.1	1.0
4	Matt Geiger	48	11.5	52.4	0.0	0.0	1.3	67.4	1.0	1.5	0.3	0.3	0.4	0.8

It's necessary to drop these highly correlated columns because they can't bring additional useful information to the model, instead, they will only increase model complexity and errors.

Separating Features & Y variable from the processed dataset

Please note to replace the dataframe below with the new dataframe created after removing highly correlated features

In [51]:
```
# Split data into features and labels
nba_new_X = nba_X.drop(columns=['Name'])
nba_new_y = nba_y
```

2.2.4 Apply the following pre-processing steps:[5 points]

1) Use OrdinalEncoding to encode the label in the dataset (male & female)

2) Convert the label from a Pandas series to a Numpy (m x 1) vector. If you don't do this, it may cause problems when implementing the logistic regression model.

3)Split the dataset into training (60%), validation (20%), and test (20%) sets.

4) Standardize the columns in the feature matrices. To avoid information leakage, learn the standardization parameters from training, and then apply training, validation and test dataset.

5) Add a column of ones to the feature matrices of train, validation and test dataset. This is a common trick so that we can learn a coefficient for the bias term of a linear model.

In [52]:
```
# Your code here
nba_new_y = np.array(nba_new_y).reshape(-1,1)
# Split the dataset into training (60%), validation (20%), and test (20%) sets
X_dev, X_test, y_dev, y_test = train_test_split(nba_new_X, nba_new_y, test_size=0.2, random_sta
X_train, X_val, y_train, y_val = train_test_split(X_dev, y_dev, test_size=0.25, random_state=0)
# Standardize the columns in the feature matrices
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)   # Fit and transform scalar on X_train
X_val = scaler.transform(X_val)           # Transform X_val
X_test = scaler.transform(X_test)         # Transform X_test
# Add a column of ones to the feature matrices
X_train = np.hstack([np.ones((X_train.shape[0], 1)), X_train])
X_val = np.hstack([np.ones((X_val.shape[0], 1)), X_val])
X_test = np.hstack([np.ones((X_test.shape[0], 1)), X_test])

print(X_train[:5], '\n\n', y_train[:5])
```

```
[[ 1.         -0.49124679 -1.3013552   0.12284448 -0.64876571 -1.1886794
  -0.92305854 -1.38055057 -0.40371009 -0.90206315 -0.90682315 -0.787819
  -0.64101233 -0.94568598]
 [ 1.         -0.96746373 -0.2913292  -1.98316695  2.25933564  0.68906735
  -0.6160512   1.48037456 -0.67030797 -0.31764835 -0.70938161 -0.30245005
  -0.1773381  -0.94568598]
 [ 1.          0.16355149 -0.82676467 -0.02640043 -0.64876571 -1.1886794
  -0.71838698 -0.03591576 -0.80360691 -0.975115    0.34363997 -0.54513453
  -0.40917521 -0.12355782]
 [ 1.          0.04449726 -0.42518807  0.70324133 -0.64876571 -1.1886794
   0.10029925 -1.2851864   0.92927929  0.04761091 -0.51194006 -0.54513453
   0.51817326  0.01346354]
 [ 1.         -0.66982814  0.12241639 -1.20377689  0.93747139  0.88869491
  -0.20670809  0.68885194 -0.80360691 -0.60985575  0.87015076  0.91097232
  -0.87284945 -0.12355782]]

 [[0.]
 [0.]
 [1.]
 [1.]
 [0.]]
```

Implement Logistic Regression

We will now implement logistic regression with L2 regularization. Given an (m x n) feature matrix X, an (m x 1) label vector y, and an (n x 1) weight vector w, the hypothesis function for logistic regression is:

$$y = \sigma(Xw)$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$, i.e. the sigmoid function. This function scales the prediction to be a probability between 0 and 1, and can then be thresholded to get a discrete class prediction.

Just as with linear regression, our objective in logistic regression is to learn the weights w which best fit the data. For L2-regularized logistic regression, we find an optimal w to minimize the following loss function:

$$\min_{w} \; -y^T \log(\sigma(Xw)) \; - \; (\mathbf{1} - y)^T \log(\mathbf{1} - \sigma(Xw)) \; + \; \alpha \|w\|_2^2$$

Unlike linear regression, however, logistic regression has no closed-form solution for the optimal w. So, we will use gradient descent to find the optimal w. The (n x 1) gradient vector g for the loss function above is:

$$g = X^T\Big(\sigma(Xw) - y\Big) + 2\alpha w$$

Below is pseudocode for gradient descent to find the optimal w. You should first initialize w (e.g. to a (n x 1) zero vector). Then, for some number of epochs t, you should update w with $w - \eta g$, where η is the learning rate and g is the gradient. You can learn more about gradient descent here (https://www.coursera.org/lecture/machine-learning/gradient-descent-8SpIM).

$w = \mathbf{0}$

for $i = 1, 2, \ldots, t$

    $w = w - \eta g$

A LogisticRegression class with five methods: train, predict, calculate_loss, calculate_gradient, and calculate_sigmoid has been implemented for you below.

In [53]:
```
class LogisticRegression():
    '''
    Logistic regression model with L2 regularization.

    Attributes
    ----------
    alpha: regularization parameter
    t: number of epochs to run gradient descent
    eta: learning rate for gradient descent
    w: (n x 1) weight vector
    '''

    def __init__(self, alpha=0, t=100, eta=1e-3):
        self.alpha = alpha
        self.t = t
        self.eta = eta
        self.w = None

    def train(self, X, y):
        '''Trains logistic regression model using gradient descent
        (sets w to its optimal value).

        Parameters
        ----------
        X : (m x n) feature matrix
        y: (m x 1) label vector

        Returns
        -------
        losses: (t x 1) vector of losses at each epoch of gradient descent
        '''

        loss = list()
        self.w = np.zeros((X.shape[1],1))
        for i in range(self.t):
            self.w = self.w - (self.eta * self.calculate_gradient(X, y))
            loss.append(self.calculate_loss(X, y))
        return loss

    def predict(self, X):
        '''Predicts on X using trained model. Make sure to threshold
        the predicted probability to return a 0 or 1 prediction.

        Parameters
        ----------
        X : (m x n) feature matrix

        Returns
        -------
        y_pred: (m x 1) 0/1 prediction vector
        '''
        y_pred = self.calculate_sigmoid(X.dot(self.w))
        y_pred[y_pred >= 0.5] = 1
        y_pred[y_pred < 0.5] = 0
        return y_pred

    def calculate_loss(self, X, y):
        '''Calculates the logistic regression loss using X, y, w,
        and alpha. Useful as a helper function for train().

        Parameters
        ----------
        X : (m x n) feature matrix
        y: (m x 1) label vector
```

```
        Returns
        -------
        loss: (scalar) logistic regression loss
        '''
        return -y.T.dot(np.log(self.calculate_sigmoid(X.dot(self.w)))) - (1-y).T.dot(np.log(1-s

    def calculate_gradient(self, X, y):
        '''Calculates the gradient of the logistic regression loss
        using X, y, w, and alpha. Useful as a helper function
        for train().

        Parameters
        ----------
        X : (m x n) feature matrix
        y: (m x 1) label vector

        Returns
        -------
        gradient: (n x 1) gradient vector for logistic regression loss
        '''
        return X.T.dot(self.calculate_sigmoid( X.dot(self.w)) - y) + 2*self.alpha*self.w


    def calculate_sigmoid(self, x):
        '''Calculates the sigmoid function on each element in vector x.
        Useful as a helper function for predict(), calculate_loss(),
        and calculate_gradient().

        Parameters
        ----------
        x: (m x 1) vector

        Returns
        -------
        sigmoid_x: (m x 1) vector of sigmoid on each element in x
        '''
        return (1)/(1 + np.exp(-x.astype('float')))
```

2.2.6 Plot Loss over Epoch and Search the space randomly to find best hyperparameters.[6 points]

A: Using your implementation above, train a logistic regression model **(alpha=0, t=100, eta=1e-3)** on the voice recognition training data. Plot the training loss over epochs. Make sure to label your axes. You should see the loss decreasing and start to converge.[2 points]

B: Using **alpha between (0,1), eta between(0, 0.001) and t between (0, 100)[3 points]**, find the best hyperparameters for LogisticRegression. You can randomly search the space 20 times to find the best hyperparameters.

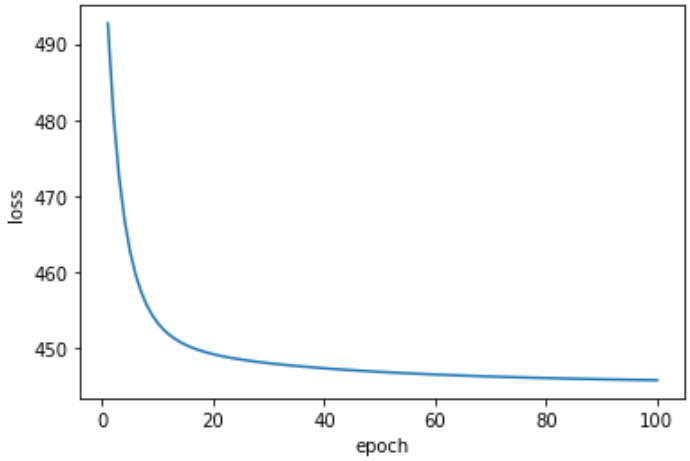
C. Compare accuracy on the test dataset for both the scenarios.[1 point]

In [54]:
```
# Your code here
# Part A
model_a = LogisticRegression()
loss = model_a.train(X_train, y_train)
loss = [l for i in range(len(loss)) for l in loss[i][0]]
epochs = [i for i in range(1, 101)]
print(min(loss))
```

```
445.84320968850824
```

In [55]:
```
plt.plot(epochs, loss)
plt.xlabel('epoch')
plt.ylabel('loss')
```

Out[55]:
```
Text(0, 0.5, 'loss')
```



In [70]:
```
# Part B and C
alphas = [np.random.rand() for i in range(20)]
etas = [0.001*np.random.rand() for i in range(20)]
ts = np.random.randint(0, 100, 20)

losses = []
models = []
for i in range(20):
    model = LogisticRegression(alphas[i], ts[i], etas[i])
    loss = model.train(X_train, y_train)
    loss = [l for i in range(len(loss)) for l in loss[i][0]]
    min_loss = min(loss)
    losses.append(min_loss)
    models.append(model)

min_l = min(losses)
i = losses.index(min_l)
best_model = models[i]
print(min_l, best_model.alpha, best_model.t, best_model.eta)
```

```
446.18245022738836 0.08506680931448873 99 0.0008544600280930272
```

In my parameter solution space, loss is minimized at 446.18 when alpha = 0.085, t = 99, eta = 0.00085.

In [75]:
```
# compare accuracy
# A: alpha=0, t=100, eta=1e-3
prediction_a = model_a.predict(X_test)
accuracy_a = accuracy_score(y_test, prediction_a)
prediction = best_model.predict(X_test)
accuracy = accuracy_score(y_test, prediction)
print(accuracy_a, accuracy, (accuracy-accuracy_a)/accuracy)
```

```
0.7164179104477612 0.7201492537313433 0.005181347150259595
```

Accuracy score for model with alpha=0, t=100, eta=1e-3 is 0.716, and that of the model with alpha=0.085, t=99, eta=85e-5 is

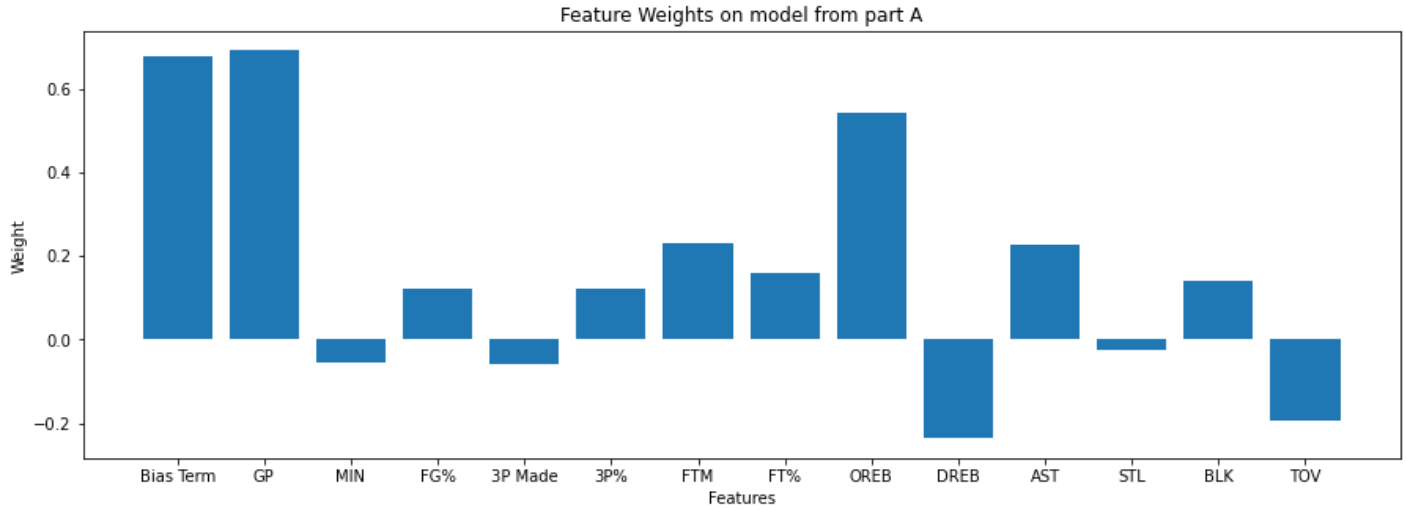
0.720. The accuracy improved 0.5% with hyperparameter searching.

Feature Importance

2.2.7 Interpret your trained model using a bar chart of the model weights. Make sure to label the bars (x-axis) and don't forget the bias term![2 points]

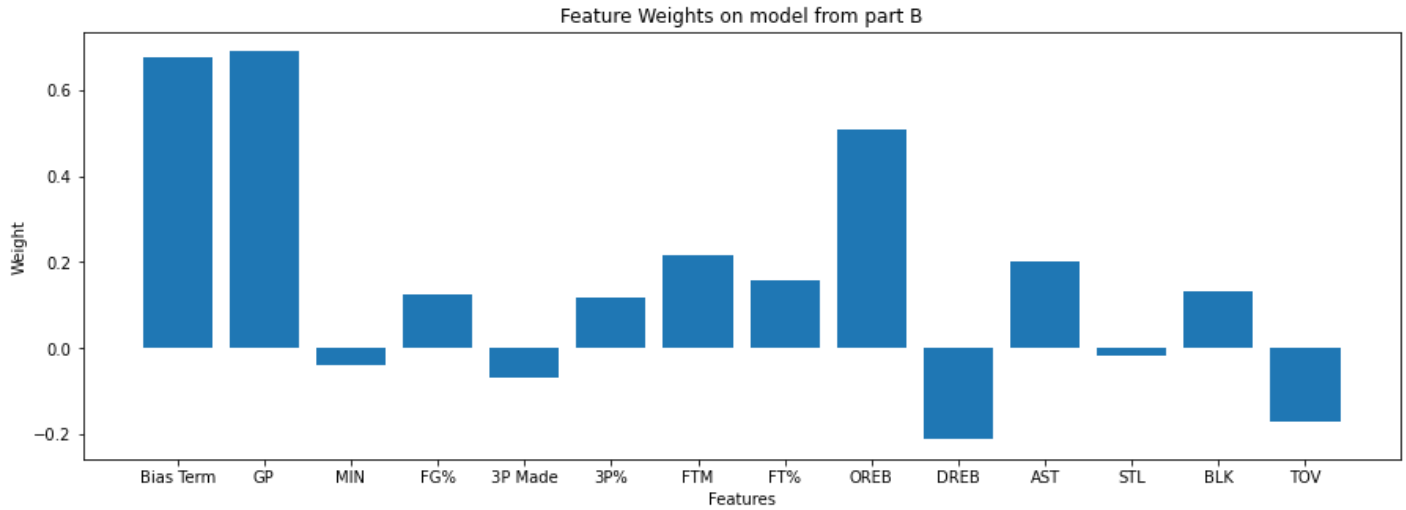
In [86]:
```
# Your code here
w = [w for i in range(len(model_a.w)) for w in model_a.w[i]]
w_names = ['Bias Term', 'GP', 'MIN', 'FG%', '3P Made', '3P%', 'FTM', 'FT%', 'OREB', 'DREB', 'AS
plt.figure(figsize=(15, 5))
plt.bar(w_names, w)
plt.xlabel('Features')
plt.ylabel('Weight')
plt.title('Feature Weights on model from part A')
```

Out[86]:
```
Text(0.5, 1.0, 'Feature Weights on model from part A')
```



In [87]:
```
w2 = [w for i in range(len(best_model.w)) for w in best_model.w[i]]
plt.figure(figsize=(15, 5))
plt.bar(w_names, w2)
plt.xlabel('Features')
plt.ylabel('Weight')
plt.title('Feature Weights on model from part B')
```

Out[87]:
```
Text(0.5, 1.0, 'Feature Weights on model from part B')
```



Both models have very similiar pattern on feature weights distribution.
Games played (GP), free throw made (FTM), assists (AST), and defensive rebound (DREB) are dominate features that predicts if a player's career is longer than 5 years or not, where the first 3 have positive wieghts, and the last one has negative weight. For example, the number of games a player plays largely predicts positively if his career is longer than 5 years.

Part 3: Support Vector Machines

In this part, we will be using a breast cancer dataset for classification.

Given 30 continuous features describing the nuclei of cells in a digitized image of a fine needle aspirate (FNA) of a breast mass, we will train SVM models to classify each sample as benign (B) or malignant (M).

In [88]:
```
cancer_df = pd.read_csv('breast-cancer.csv')
cancer_df = cancer_df.drop(columns=['id', 'Unnamed: 32'])
cancer_df
```

Out[88]:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_me
0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30(
1	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08(
2	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19
3	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24
4	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19(
...	...	...	...	...	...	...	...	
564	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24
565	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14
566	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09
567	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35
568	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00(

569 rows × 31 columns

In [89]:
```
# Split data into features and labels

cancer_X = cancer_df.drop(columns=['diagnosis'])
cancer_y = cancer_df['diagnosis']
```

The following pre-processing steps have been applied to the breast cancer dataset in the next cell:

1. Encode the categorical label as 0 (B) or 1 (M).
2. Convert the label from a Pandas series to a Numpy (m x 1) vector. If you don't do this, it may cause problems when implementing the logistic regression model (certain broadcasting operations may fail unexpectedly).
3. Split the dataset into training (60%), validation (20%), and test (20%) sets.
4. Standardize the columns in the feature matrices cancer_X_train, cancer_X_val, and cancer_X_test to have zero mean and unit variance. To avoid information leakage, learn the standardization parameters (mean, variance) from cancer_X_train, and apply it to cancer_X_train, cancer_X_val, and cancer_X_test.
5. Add a column of ones to the feature matrices cancer_X_train, cancer_X_val, and cancer_X_test. This is a common trick so that we can learn a coefficient for the bias term of a linear model.

In [96]:
```
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder

cancer_df['diagnosis'] = cancer_df.diagnosis.astype("category").cat.codes
cancer_y_enc = cancer_df['diagnosis'].to_numpy()
cancer_y_enc = cancer_y_enc.reshape(cancer_y_enc.shape[0],1)
print(cancer_y_enc.shape)
print(type(cancer_y_enc))


cancer_X_dev, cancer_X_test, cancer_y_dev, cancer_y_test = train_test_split(cancer_X, cancer_y_
cancer_X_train, cancer_X_val, cancer_y_train, cancer_y_val = train_test_split(cancer_X_dev, car


scaler = StandardScaler()
cancer_X_train = scaler.fit_transform(cancer_X_train)
cancer_X_val = scaler.transform(cancer_X_val)
cancer_X_test = scaler.transform(cancer_X_test)


cancer_X_train = np.hstack([np.ones((cancer_X_train.shape[0], 1)), cancer_X_train])
cancer_X_val = np.hstack([np.ones((cancer_X_val.shape[0], 1)), cancer_X_val])
cancer_X_test = np.hstack([np.ones((cancer_X_test.shape[0], 1)), cancer_X_test])
```

```
(569, 1)
<class 'numpy.ndarray'>
```

Train Primal SVM

3.1 Train a primal SVM (with default parameters) on the breast cancer training data. Make predictions and report the accuracy on the training, validation, and test sets.[5 points]

In [99]:
```
# Your code here
primal = LinearSVC()
primal.fit(cancer_X_train, cancer_y_train.ravel())
pred_train = primal.predict(cancer_X_train)
pred_val = primal.predict(cancer_X_val)
pred_test = primal.predict(cancer_X_test)
accuracy_train = accuracy_score(cancer_y_train, pred_train)
accuracy_val = accuracy_score(cancer_y_val, pred_val)
accuracy_test = accuracy_score(cancer_y_test, pred_test)
print(accuracy_train, accuracy_val, accuracy_test)
```

```
0.9912023460410557 0.9298245614035088 0.9473684210526315
```

The accuracy on training is 0.991, validation 0.930, and test 0.947.

Train Dual SVM

3.2 Train a dual SVM (with default parameters) on the breast cancer training data. Make predictions and report the accuracy on the training, validation, and test sets.[5 points]

In [101]:
```
# Your code here
dual = SVC()
dual.fit(cancer_X_train, cancer_y_train.ravel())
pred_train = dual.predict(cancer_X_train)
pred_val = dual.predict(cancer_X_val)
pred_test = dual.predict(cancer_X_test)
accuracy_train = accuracy_score(cancer_y_train, pred_train)
accuracy_val = accuracy_score(cancer_y_val, pred_val)
accuracy_test = accuracy_score(cancer_y_test, pred_test)
print(accuracy_train, accuracy_val, accuracy_test)
```

```
0.9853372434017595 0.9824561403508771 0.9736842105263158
```

The accuracy on training is 0.985, validation 0.982, and test 0.974.