Lab 3 Extending Logistic Regression

By. Eli Laird, Liam Lowsley-Williams, Fernando Vazquez

In [82]:
```
import pandas as pd
import numpy as np
from scipy.special import expit
from numpy.linalg import pinv
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import plotly
import plotly.graph_objects as go
plotly.offline.init_notebook_mode()
```

Buisiness Understanding

The data set we chose consists of several thousand mushroom records drawn from The Audubon Society Field Guide to North American Mushrooms. It includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one.

Our current use case for this dataset is to try and predict what the habitat of the mushroom would be given its physical features and related information. This use case could be beneficial for a variety of reasons, for one, potentially being able to identify where poisonous mushrooms may exist could help prevent mushroom related illness or even death. For adventurists that are exposed to these types of situations fairly frequently and who may also want to pick their own fresh mushrooms, we could use logistic regression to classify if a given mushroom may be poisonous.

Mushrooms are also very important to our ecosystem and another use case for this project is to be able to audit mushrooms quickly in order to keep up with how they are surviving in the wild. Being able to identify random or aggressive changes in mushroom vegetation and survival could prove to be extremely useful especially in identifying when we may be facing adverse climate change or pollution.

Third parties that could be potentially interested in this classification algorithm would be environmental activists, the EPA, adventurists, and any other people who could be interested in classifying mushrooms by their features.

Should the regression analysis perform well and fast enough the classification could be done on the fly, such as through an app. Users could highlighting key features and allowing them to predict the mushroom class could allow adventurists, naturists, and any curious human beings identify key characteristics regarding mushrooms growing in the wild. It could also prove to be useful in helping identify, if a pet or toddler ate a mushroom by accident, whether they need to seek immediate medical attention or call poison control. “The Guide clearly states that there is no simple rule for determining the edibility of a mushroom”, thus providing a means of classification for mushrooms would be very beneficial.

Data Preparation

In [83]:
```
column_names = ['class',
                'cap-shape',
                'cap-surface',
                'cap-color',
                'bruises',
                'odor',
                'gill-attachment',
                'gill-spacing',
                'gill-size',
                'gill-color',
                'stalk-shape',
                'stalk-root',
                'stalk-surface-above-ring',
                'stalk-surface-below-ring',
                'stalk-color-above-ring',
                'stalk-color-below-ring',
                'veil-type',
                'veil-color',
                'ring-number',
                'ring-type',
                'spore-print-color',
                'population',
                'habitat']

mushrooms = pd.read_csv('data/agaricus-lepiota.data', header=None, names=colum
n_names)
```

In [84]:
```
mushrooms.describe()
```

Out[84]:

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	st surfa bel
count	8124	8124	8124	8124	8124	8124	8124	8124	8124	8124	...	8
unique	2	6	4	10	2	9	2	2	2	12	...	
top	e	x	y	n	f	n	f	c	b	b	...	
freq	4208	3656	3244	2284	4748	3528	7914	6812	5612	1728	...	4

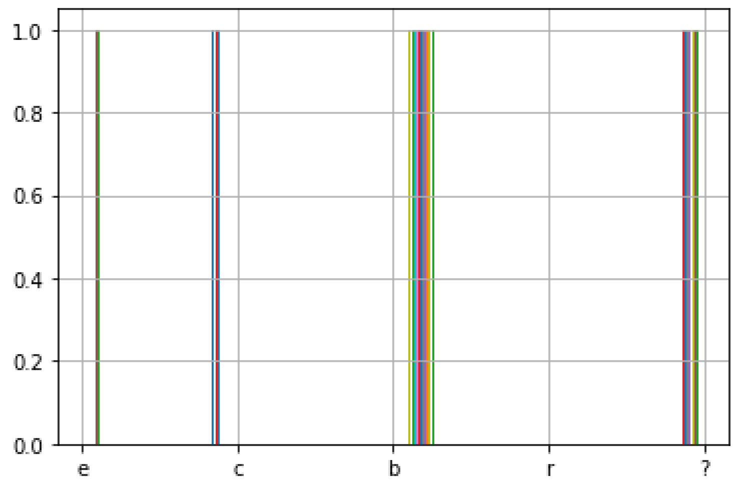
4 rows × 23 columns

In [85]:
```
mushrooms['stalk-root'].unique()
```

Out[85]:
```
array(['e', 'c', 'b', 'r', '?'], dtype=object)
```

In [14]:
```
mushrooms['stalk-root'].hist()
```

Out[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1e870e5af60>
```



According to the data set documentation, "?" represents missing data. Due to the high number of missing values for this feature, we decided to remove "stalk-root" from our feature set.

In [86]:
```
mushrooms.drop(['stalk-root'], inplace=True, axis=1)
```

In [87]:
```
# extract y values from data set
shroomsy = mushrooms["habitat"]
shroomsy
```

Out[87]:
```
0       u
1       g
2       m
3       u
4       g
5       g
6       m
7       m
8       g
9       m
10      g
11      m
12      g
13      u
14      g
15      u
16      g
17      g
18      u
19      u
20      m
21      g
22      m
23      m
24      m
25      g
26      m
27      m
28      u
29      d
       ..
8094    g
8095    d
8096    g
8097    l
8098    d
8099    g
8100    l
8101    p
8102    l
8103    l
8104    l
8105    l
8106    l
8107    l
8108    l
8109    g
8110    l
8111    g
8112    l
8113    d
8114    d
8115    l
8116    l
8117    d
8118    d
8119    l
```

```
8120    l
8121    l
8122    l
8123    l
Name: habitat, Length: 8124, dtype: object
```

In [88]:
```
# extract one-hot encoding of X values from data set
shroomsX = pd.get_dummies(mushrooms.drop(['habitat'],axis=1))
shroomsX
```

Out[88]:

	class_e	class_p	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	sur
0	0	1	0	0	0	0	0	1	0	
1	1	0	0	0	0	0	0	1	0	
2	1	0	1	0	0	0	0	0	0	
3	0	1	0	0	0	0	0	1	0	
4	1	0	0	0	0	0	0	1	0	
5	1	0	0	0	0	0	0	1	0	
6	1	0	1	0	0	0	0	0	0	
7	1	0	1	0	0	0	0	0	0	
8	0	1	0	0	0	0	0	1	0	
9	1	0	1	0	0	0	0	0	0	
10	1	0	0	0	0	0	0	1	0	
11	1	0	0	0	0	0	0	1	0	
12	1	0	1	0	0	0	0	0	0	
13	0	1	0	0	0	0	0	1	0	
14	1	0	0	0	0	0	0	1	1	
15	1	0	0	0	0	0	1	0	1	
16	1	0	0	0	1	0	0	0	1	
17	0	1	0	0	0	0	0	1	0	
18	0	1	0	0	0	0	0	1	0	
19	0	1	0	0	0	0	0	1	0	
20	1	0	1	0	0	0	0	0	0	
21	0	1	0	0	0	0	0	1	0	
22	1	0	1	0	0	0	0	0	0	
23	1	0	1	0	0	0	0	0	0	
24	1	0	1	0	0	0	0	0	0	
25	0	1	0	0	1	0	0	0	0	
26	1	0	0	0	0	0	0	1	0	
27	1	0	0	0	0	0	0	1	0	
28	1	0	0	0	1	0	0	0	1	
29	1	0	0	0	0	0	0	1	0	
...	...	...	...	...	...	...	...	...	...	
8094	1	0	1	0	0	0	0	0	0	
8095	0	1	0	0	0	0	0	1	0	
8096	1	0	0	0	0	1	0	0	1	

	class_e	class_p	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	sur
8097	0	1	0	0	0	1	0	0	0	
8098	0	1	0	0	0	1	0	0	0	
8099	1	0	0	0	0	1	0	0	1	
8100	1	0	0	0	1	0	0	0	0	
8101	0	1	0	0	0	1	0	0	0	
8102	1	0	0	0	0	0	0	1	0	
8103	1	0	0	0	0	1	0	0	0	
8104	1	0	0	0	0	1	0	0	0	
8105	1	0	0	0	0	1	0	0	0	
8106	1	0	0	0	0	1	0	0	0	
8107	1	0	0	0	0	0	0	1	0	
8108	0	1	0	0	0	1	0	0	0	
8109	1	0	1	0	0	0	0	0	0	
8110	1	0	0	0	0	0	0	1	0	
8111	1	0	0	0	0	1	0	0	0	
8112	1	0	0	0	0	1	0	0	0	
8113	0	1	0	0	0	1	0	0	0	
8114	0	1	0	0	1	0	0	0	0	
8115	1	0	0	0	0	0	0	1	0	
8116	0	1	0	0	0	1	0	0	0	
8117	0	1	0	0	0	1	0	0	0	
8118	0	1	0	0	0	1	0	0	0	
8119	1	0	0	0	0	1	0	0	0	
8120	1	0	0	0	0	0	0	1	0	
8121	1	0	0	0	1	0	0	0	0	
8122	0	1	0	0	0	1	0	0	0	
8123	1	0	0	0	0	0	0	1	0	

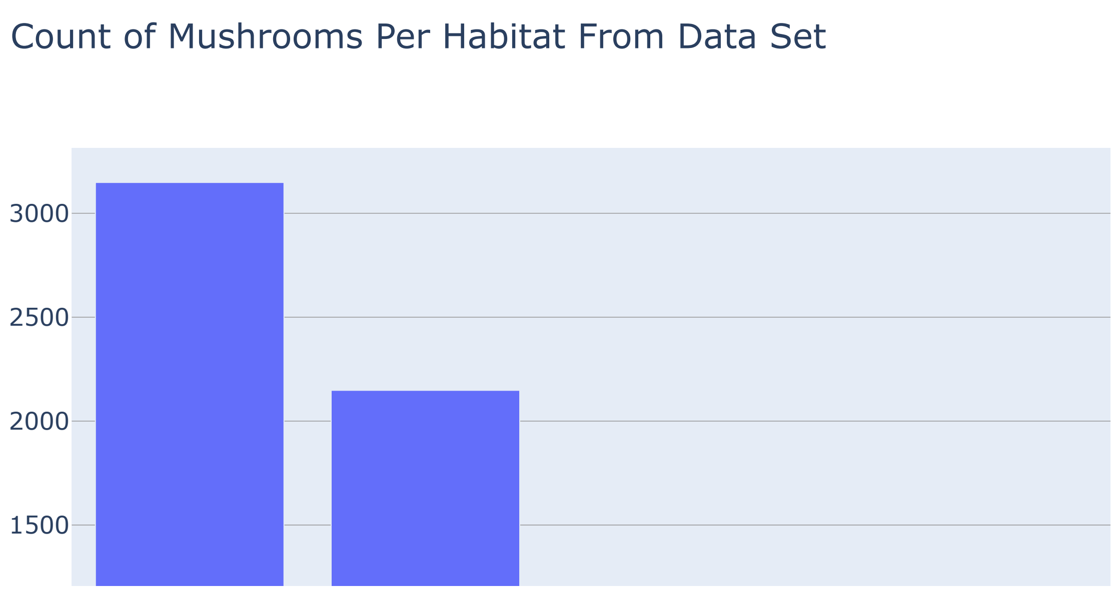
8124 rows × 107 columns

In [89]:
```
graph_y = shroomsy.value_counts()

fig = go.Figure([go.Bar(x=graph_y.index, y=graph_y.values)])

fig.update_layout(
    title = 'Count of Mushrooms Per Habitat From Data Set',
    xaxis = dict(
        tickmode = 'array',
        tickvals = ['d',     'g',       'p',     'l',      'u',     'm',
'w'],
        ticktext = ['Woods', 'Grasses', 'Paths', 'Leaves', 'Urban', 'Meadows',
'Waste']
    )
)


fig.show()
```



Implementation

Function Parameters

- *Eta*
- *Iterations*
- *C*
    - default=0
- *Optimization* (Required)
    - 'bgd' = Batch Gradient Descent
    - 'sgd' = Stochastic Gradient Descent
    - 'newtons' = Newton's Method
- *Regularization*
    - L1 (Lasso) -> 'lasso'
    - L2 (Ridge) -> 'ridge'
    - Both -> 'elastic_net'

Default:

- LogisticRegression(optimization='bgd', eta = 0.01, iterations=20, regularization='ridge', c=0)

In [90]:
```
class BinaryLogisticRegressionBase:
    # private:
    def __init__(self, optimization='bgd', eta = 0.01, iterations=20, regulari
zation='ridge', c=0):
        self.eta = eta
        self.iters = iterations
        self.opt = optimization
        self.reg = regularization
        self.c = c
        # internally we will store the weights as self.w_ to keep with sklearn
conventions

    def __str__(self):
        return 'Base Binary Logistic Regression Object, Not Trainable'

    # convenience, private and static:
    @staticmethod
    def _sigmoid(theta):
        return 1/(1+np.exp(-theta))

    @staticmethod
    def _add_bias(X):
        return np.hstack((np.ones((X.shape[0],1)),X)) # add bias term

    # public:
    def predict_proba(self,X,add_bias=True):
        # add bias term if requested
        Xb = self._add_bias(X) if add_bias else X
        return self._sigmoid(Xb @ self.w_) # return the probability y=1

    def predict(self,X):
        return (self.predict_proba(X)>0.5) #return the actual prediction
```

In [91]:
```
class BinaryLogisticRegression(BinaryLogisticRegressionBase):
    #private:
    def __str__(self):
        if(hasattr(self,'w_')):
            return 'Binary Logistic Regression Object with coefficients:\n'+ s
tr(self.w_) # is we have trained the object
        else:
            return 'Untrained Binary Logistic Regression Object'

    #optimization methods
    def _get_gradient(self, X, y):

        gradient = None
        if self.opt == 'bgd': gradient = self._batch_gradient_descent
        elif self.opt == 'sgd': gradient = self._stochastic_gradient_descent
        elif self.opt == 'newton': gradient = self._newtons_method
        elif self.opt == 'one_hessian': gradient = self._one_step_hessian

        return gradient(X,y)

    def _batch_gradient_descent(self,X,y):
        ydiff = y-self.predict_proba(X,add_bias=False).ravel() # get y differe
nce
        gradient = np.mean(X * ydiff[:,np.newaxis], axis=0) # make ydiff a col
umn vector and multiply through
        gradient = gradient.reshape(self.w_.shape)
        gradient[1:] += self.c * self._get_reg_gradient()

        return gradient

    def _stochastic_gradient_descent(self,X,y):
       # idx = int(np.random.rand()*len(y)) # grab random instance\
        idx = np.random.randint(len(y))
        ydiff = y[idx]-self.predict_proba(X[idx],add_bias=False) # get y diffe
rence (now scalar)
        gradient = X[idx] * ydiff[:,np.newaxis] # make ydiff a column vector a
nd multiply through

        gradient = gradient.reshape(self.w_.shape)
        gradient[1:] += self.c * self._get_reg_gradient()

        return gradient

    def _newtons_method(self,X,y):
        sigmoid_z = (sigma1*X + sigma2).astype("float_")
        sigmoid = 1.0/(1.0 + np.exp(-z))
        return np.sum(y * np.log(sigmoid) + (1 - y) * np.log(1 - sigmoid))

    def _one_step_hessian(self, X, y):
        g = self.predict_proba(X,add_bias=False).ravel() # get sigmoid value f
or all classes
        hessian = X.T @ np.diag(g*(1-g)) @ X - 2 * self.c  # calculate the hes
sian

        ydiff = y-g # get y difference
        gradient = np.sum(X * ydiff[:,np.newaxis], axis=0) # make ydiff a colu
```

```
mn vector and multiply through
        gradient = gradient.reshape(self.w_.shape)
        gradient[1:] +=  self._get_reg_gradient()

        return pinv(hessian) @ gradient

    @staticmethod
    def _sigmoid(theta):
        # increase stability, redefine sigmoid operation
        return expit(theta) #1/(1+np.exp(-theta))

    #regularization methods
    def _get_reg_gradient(self):
        if self.reg == 'ridge': return -2 * self.w_[1:]
        elif self.reg == 'lasso': return np.sign(self.w_[1:])
        elif self.reg == 'elastic_net': return -2 * self.w_[1:] + np.sign(self
.w_[1:])


    # public:
    def fit(self, X, y):
        Xb = self._add_bias(X) # add bias term
        num_samples, num_features = Xb.shape

        self.w_ = np.zeros((num_features,1)) # init weight vector to zeros

        # for as many as the max iterations
        for _ in range(self.iters):
            gradient = self._get_gradient(Xb,y)
            self.w_ += gradient*self.eta # multiply by learning rate
```

In [92]:

```
class VectorBinaryLogisticRegression(BinaryLogisticRegression):
    # inherit from our previous class to get same functionality
    @staticmethod
    def _sigmoid(theta):
        # increase stability, redefine sigmoid operation
        return expit(theta) #1/(1+np.exp(-theta))

    # but overwrite the gradient calculation
    def _get_gradient(self,X,y):
        ydiff = y-self.predict_proba(X,add_bias=False).ravel() # get y differe
nce
        gradient = np.mean(X * ydiff[:,np.newaxis], axis=0) # make ydiff a col
umn vector and multiply through

        return gradient.reshape(self.w_.shape)
```

Logistic Regression Class

In [93]:
```
class LogisticRegression:

    def __init__(self, optimization='bgd', eta = 0.01, iterations=20, regulari
zation='ridge', c=0):

        self.eta = eta
        self.iters = iterations
        self.opt = optimization
        self.reg = regularization
        self.class_encodings_ = {}
        self.c = c


    def __str__(self):
        if(hasattr(self,'w_')):
            return 'MultiClass Logistic Regression Object with coefficients:\n
'+ str(self.w_) # is we have trained the object
        else:
            return 'Untrained MultiClass Logistic Regression Object'


    def fit(self,X,y):
        num_samples, num_features = X.shape
        self.unique_ = np.unique(y) # get each unique class value
        num_unique_classes = len(self.unique_)
        self.classifiers_ = [] # will fill this array with binary classifiers

        for i,yval in enumerate(self.unique_): # for each unique value
            self.class_encodings_[yval] = i
            y_binary = (y==yval) # create a binary problem
            # train the binary classifier for this class
            blr = BinaryLogisticRegression(self.opt, self.eta, self.iters, sel
f.reg, self.c )
            blr.fit(X,y_binary)
            # add the trained classifier to the list
            self.classifiers_.append(blr)

        # save all the weights into one matrix, separate column for each class
        self.w_ = np.hstack([x.w_ for x in self.classifiers_]).T

    def predict_proba(self,X):
        probs = []
        for blr in self.classifiers_:
            probs.append(blr.predict_proba(X)) # get probability for each clas
sifier

        return np.hstack(probs) # make into single matrix

    def predict(self,X):
        return np.argmax(self.predict_proba(X),axis=1) # take argmax along row


lr = LogisticRegression('bgd',0.01, 100, 'ridge')
print(lr)
```

```
Untrained MultiClass Logistic Regression Object
```

An 80/20 split of the data is appropriate for our dataset because at 80% we can assume that we captured the majority of the variability in the data.

In [94]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(shroomsX, shroomsy, test_s
ize=.20, random_state=42)

X_train = X_train.to_numpy()
X_test = X_test.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

print("X_train type:", type(X_train))
print("X_test type:", type(X_test))
print("y_train type:", type(y_train))
print("y_test type:", type(y_test))
```

```
X_train type: <class 'numpy.ndarray'>
X_test type: <class 'numpy.ndarray'>
y_train type: <class 'numpy.ndarray'>
y_test type: <class 'numpy.ndarray'>
```

Evaluate on training data with the Iris data set

In [95]:
```
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
ds = load_iris()
X = ds.data
y = ds.target

lr = LogisticRegression(optimization='bgd',eta=0.1, iterations=500, c=.01)
lr.fit(X,y)
yprobs = lr.predict_proba(X)

yhat = lr.predict(X)
print("YHat", yhat)
print('Accuracy of: ',accuracy_score(y,yhat))
```

```
YHat [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1 1 2 1 1 1 2 1 2 1
 1 1 1 2 1 1 1 1 1 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
Accuracy of:  0.94
```

Evaluate on training data with the our data set

In [96]:
```
lr = LogisticRegression(optimization='bgd',eta=0.1, regularization='ridge', iterations=500)
lr.fit(X_train, y_train)
pred = lr.predict(X_train)

encode = lambda x: lr.class_encodings_[x]

y_train_encode = np.array(list(map(encode, y_train)))

train_mse = accuracy_score(y_train_encode, pred)

print("Training MSE: ", train_mse)
```

```
Training MSE:  0.6377904292968148
```

Evaluate on test data with the our data set

In [113]:
```
lr = LogisticRegression(optimization='bgd',eta=0.1, regularization='ridge', iterations=500)
lr.fit(X_test, y_test)
#probas = lr.predict_proba(X_train)
pred = lr.predict(X_test)

encode = lambda x: lr.class_encodings_[x]

y_test_encode = np.array(list(map(encode, y_test)))

test_mse = accuracy_score(y_test_encode, pred)

print("Testing MSE: ", test_mse)
```

```
Testing MSE:  0.6504615384615384
```

In [97]:
```
print(y_train)
print(y_train_encode)
print(lr.class_encodings_)
```

```
['d' 'p' 'l' ... 'p' 'p' 'g']
[0 4 2 ... 4 4 1]
{'d': 0, 'g': 1, 'l': 2, 'm': 3, 'p': 4, 'u': 5, 'w': 6}
```

In the following cells we try to find the best optimization technique and regularization term to achive the best performance for our test set. Our selection of parameters involves going through each an every combination for an optimization technique and a regularization term. There isnt any data snopping in our technique.

In [98]:
```
valArr=[]
startNum=0.000001
for x in range(0,6):

    valArr.append(startNum)
    startNum*=10
print(valArr)

resultsList=[]
for e in valArr:
    print(e)
    lr = LogisticRegression(optimization='bgd',eta=e, regularization='ridge',
iterations=300)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".fo
rmat(train_mse,e,"bgd","ridge"))
    resultsList.append([train_mse,e,"bgd","ridge"])

    lr = LogisticRegression(optimization='bgd',eta=e, regularization='lasso',
iterations=300)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".fo
rmat(train_mse,e,"bgd","lasso"))
    resultsList.append([train_mse,e,"bgd","lasso"])

    lr = LogisticRegression(optimization='bgd',eta=e, regularization='elastic_
net', iterations=300)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".fo
rmat(train_mse,e,"bgd","elastic_net"))
    resultsList.append([train_mse,e,"bgd","elastic_net"])
```

```
[1e-06, 9.999999999999999e-06, 9.999999999999999e-05, 0.001, 0.01, 0.1]
1e-06
Training MSE: 0.38667487305739345, Eta: 1e-06, optimization: bgd, regularizat
ion: ridge
Training MSE: 0.38667487305739345, Eta: 1e-06, optimization: bgd, regularizat
ion: lasso
Training MSE: 0.38667487305739345, Eta: 1e-06, optimization: bgd, regularizat
ion: elastic_net
9.999999999999999e-06
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-06, optimization:
bgd, regularization: ridge
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-06, optimization:
bgd, regularization: lasso
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-06, optimization:
bgd, regularization: elastic_net
9.999999999999999e-05
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-05, optimization:
bgd, regularization: ridge
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-05, optimization:
bgd, regularization: lasso
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-05, optimization:
bgd, regularization: elastic_net
0.001
Training MSE: 0.39021387905831667, Eta: 0.001, optimization: bgd, regularizat
ion: ridge
Training MSE: 0.39021387905831667, Eta: 0.001, optimization: bgd, regularizat
ion: lasso
Training MSE: 0.39021387905831667, Eta: 0.001, optimization: bgd, regularizat
ion: elastic_net
0.01
Training MSE: 0.5371595630096938, Eta: 0.01, optimization: bgd, regularizatio
n: ridge
Training MSE: 0.5371595630096938, Eta: 0.01, optimization: bgd, regularizatio
n: lasso
Training MSE: 0.5371595630096938, Eta: 0.01, optimization: bgd, regularizatio
n: elastic_net
0.1
Training MSE: 0.6279427604246807, Eta: 0.1, optimization: bgd, regularizatio
n: ridge
Training MSE: 0.6279427604246807, Eta: 0.1, optimization: bgd, regularizatio
n: lasso
Training MSE: 0.6279427604246807, Eta: 0.1, optimization: bgd, regularizatio
n: elastic_net
```

In [99]:
```
resultsListsgd=[]
for e in valArr:
    print(e)
    lr = LogisticRegression(optimization='sgd',eta=e, regularization='ridge', iterations=300)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".format(train_mse,e,"sgd","ridge"))
    resultsListsgd.append([train_mse,e,"sgd","ridge"])

    lr = LogisticRegression(optimization='sgd',eta=e, regularization='lasso', iterations=300)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".format(train_mse,e,"sgd","lasso"))
    resultsListsgd.append([train_mse,e,"sgd","lasso"])

    lr = LogisticRegression(optimization='sgd',eta=e, regularization='elastic_net', iterations=300)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".format(train_mse,e,"sgd","elastic_net"))
    resultsListsgd.append([train_mse,e,"sgd","elastic_net"])
```

```
1e-06
Training MSE: 0.38667487305739345, Eta: 1e-06, optimization: sgd, regularizat
ion: ridge
Training MSE: 0.38667487305739345, Eta: 1e-06, optimization: sgd, regularizat
ion: lasso
Training MSE: 0.38667487305739345, Eta: 1e-06, optimization: sgd, regularizat
ion: elastic_net
9.999999999999999e-06
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-06, optimization:
sgd, regularization: ridge
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-06, optimization:
sgd, regularization: lasso
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-06, optimization:
sgd, regularization: elastic_net
9.999999999999999e-05
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-05, optimization:
sgd, regularization: ridge
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-05, optimization:
sgd, regularization: lasso
Training MSE: 0.38667487305739345, Eta: 9.999999999999999e-05, optimization:
sgd, regularization: elastic_net
0.001
Training MSE: 0.38667487305739345, Eta: 0.001, optimization: sgd, regularizat
ion: ridge
Training MSE: 0.4997691952608094, Eta: 0.001, optimization: sgd, regularizati
on: lasso
Training MSE: 0.38667487305739345, Eta: 0.001, optimization: sgd, regularizat
ion: elastic_net
0.01
Training MSE: 0.5476227111863363, Eta: 0.01, optimization: sgd, regularizatio
n: ridge
Training MSE: 0.5286967225727035, Eta: 0.01, optimization: sgd, regularizatio
n: lasso
Training MSE: 0.5423911370980151, Eta: 0.01, optimization: sgd, regularizatio
n: elastic_net
0.1
Training MSE: 0.6273272811201723, Eta: 0.1, optimization: sgd, regularizatio
n: ridge
Training MSE: 0.5922449607631943, Eta: 0.1, optimization: sgd, regularizatio
n: lasso
Training MSE: 0.620403138944453, Eta: 0.1, optimization: sgd, regularization:
elastic_net
```

In [100]:

```
resultsListone_hessian=[]
for e in valArr:
    print(e)
    lr = LogisticRegression(optimization='one_hessian',eta=e, regularization=
'ridge', iterations=1)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".fo
rmat(train_mse,e,"one_hessian","ridge"))
    resultsListone_hessian.append([train_mse,e,"one_hessian","ridge"])

    lr = LogisticRegression(optimization='one_hessian',eta=e, regularization=
'lasso', iterations=1)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".fo
rmat(train_mse,e,"one_hessian","lasso"))
    resultsListone_hessian.append([train_mse,e,"one_hessian","lasso"])

    lr = LogisticRegression(optimization='one_hessian',eta=e, regularization=
'elastic_net', iterations=1)
    lr.fit(X_train, y_train)
    pred = lr.predict(X_train)
    encode = lambda x: lr.class_encodings_[x]
    y_train_encode = np.array(list(map(encode, y_train)))
    train_mse = accuracy_score(y_train_encode, pred)
    print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".fo
rmat(train_mse,e,"one_hessian","elastic_net"))
    resultsListone_hessian.append([train_mse,e,"one_hessian","elastic_net"])
```

```
1e-06
Training MSE: 0.6784120633943683, Eta: 1e-06, optimization: one_hessian, regu
larization: ridge
Training MSE: 0.6784120633943683, Eta: 1e-06, optimization: one_hessian, regu
larization: lasso
Training MSE: 0.6784120633943683, Eta: 1e-06, optimization: one_hessian, regu
larization: elastic_net
9.999999999999999e-06
Training MSE: 0.6784120633943683, Eta: 9.999999999999999e-06, optimization: o
ne_hessian, regularization: ridge
Training MSE: 0.6784120633943683, Eta: 9.999999999999999e-06, optimization: o
ne_hessian, regularization: lasso
Training MSE: 0.6784120633943683, Eta: 9.999999999999999e-06, optimization: o
ne_hessian, regularization: elastic_net
9.999999999999999e-05
Training MSE: 0.6784120633943683, Eta: 9.999999999999999e-05, optimization: o
ne_hessian, regularization: ridge
Training MSE: 0.6784120633943683, Eta: 9.999999999999999e-05, optimization: o
ne_hessian, regularization: lasso
Training MSE: 0.6784120633943683, Eta: 9.999999999999999e-05, optimization: o
ne_hessian, regularization: elastic_net
0.001
Training MSE: 0.6784120633943683, Eta: 0.001, optimization: one_hessian, regu
larization: ridge
Training MSE: 0.6784120633943683, Eta: 0.001, optimization: one_hessian, regu
larization: lasso
Training MSE: 0.6784120633943683, Eta: 0.001, optimization: one_hessian, regu
larization: elastic_net
0.01
Training MSE: 0.6784120633943683, Eta: 0.01, optimization: one_hessian, regul
arization: ridge
Training MSE: 0.6784120633943683, Eta: 0.01, optimization: one_hessian, regul
arization: lasso
Training MSE: 0.6784120633943683, Eta: 0.01, optimization: one_hessian, regul
arization: elastic_net
0.1
Training MSE: 0.6784120633943683, Eta: 0.1, optimization: one_hessian, regula
rization: ridge
Training MSE: 0.6784120633943683, Eta: 0.1, optimization: one_hessian, regula
rization: lasso
Training MSE: 0.6784120633943683, Eta: 0.1, optimization: one_hessian, regula
rization: elastic_net
```

In [101]:
```
maxnum=[0]
for xlist in resultsList:
    if xlist[0]>maxnum[0]:
        maxnum=xlist
print(maxnum)
maxnumsgd=[0]
for xlist in resultsListsgd:
    if xlist[0]>maxnumsgd[0]:
        maxnumsgd=xlist
print(maxnumsgd)
maxnumHessian=[0]
for xlist in resultsListone_hessian:
    if xlist[0]>maxnumHessian[0]:
        maxnumHessian=xlist
print(maxnumHessian)
```

```
[0.6279427604246807, 0.1, 'bgd', 'ridge']
[0.6273272811201723, 0.1, 'sgd', 'ridge']
[0.6784120633943683, 1e-06, 'one_hessian', 'ridge']
```

Comparision of SK Learn's Logistic Regression and our own implementation

Our best logistic regression at 67.84% used the as parameters opitimization = 'one_hessian', regularization = 'ridge' and eta = '1e-6'. I would also like to note that all of the results for using the one_hessian optimization were the same.

Running test and train data with our Logistic Regression

In [111]:
```
lr = LogisticRegression(optimization='one_hessian',eta=1e-06, regularization=
'ridge', iterations=1)
%time lr.fit(X_train, y_train)
pred = lr.predict(X_train)
encode = lambda x: lr.class_encodings_[x]
y_train_encode = np.array(list(map(encode, y_train)))
train_mse = accuracy_score(y_train_encode, pred)
print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".format
(train_mse,e,"one_hessian","ridge"))

lr = LogisticRegression(optimization='one_hessian',eta=1e-06, regularization=
'ridge', iterations=1)
%time lr.fit(X_test, y_test)
pred = lr.predict(X_test)
encode = lambda x: lr.class_encodings_[x]
y_train_encode = np.array(list(map(encode, y_test)))
test_mse = accuracy_score(y_train_encode, pred)
print("Test MSE: {}, Eta: {}, optimization: {}, regularization: {}".format(tes
t_mse,e,"one_hessian","ridge"))
```

```
Wall time: 1.38 s
Training MSE: 0.6784120633943683, Eta: 0.1, optimization: one_hessian, regula
rization: ridge
Wall time: 118 ms
Test MSE: 0.7132307692307692, Eta: 0.1, optimization: one_hessian, regulariza
tion: ridge
```

Running test and train data with SKlearn's Logistic Regression

In [110]:
```
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression as LogisticRegressionSK

print("Run with train data")
clf = LogisticRegressionSK()
%time clf.fit(X_train, y_train)
clf.predict(X_train)
clf.predict_proba(X_train)
clf.score(X_train,y_train)

print("************)
print("Run with Test data")
clf = LogisticRegressionSK()
%time clf.fit(X_test, y_test)
clf.predict(X_test)
clf.predict_proba(X_test)
clf.score(X_test,y_test)
```

```
Run with train data

C:\Users\vazqu\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:4
32: FutureWarning:

Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silenc
e this warning.

C:\Users\vazqu\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:4
69: FutureWarning:

Default multi_class will be changed to 'auto' in 0.22. Specify the multi_clas
s option to silence this warning.


Wall time: 217 ms
Run with Test data
Wall time: 52 ms

C:\Users\vazqu\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:4
32: FutureWarning:

Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silenc
e this warning.

C:\Users\vazqu\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:4
69: FutureWarning:

Default multi_class will be changed to 'auto' in 0.22. Specify the multi_clas
s option to silence this warning.
```

Out[110]:
```
0.7206153846153847
```

SKlearn's logistic regression had a slightly higher accuracy at 68.05662% while our implementation's accuracy was 67.84120% using the training data and for the testing data the accuracies were 71.32308% for our implementation and 72.06154% for SKlearns. The most noticable difference between Sklearns implementation and ours was the time it took to run the .fit() function. Sklearn ran the function in 217ms for training and 52ms for testing while our implementation ran in 1.38sec for trai ning and 118ms for testing. These results are expected given the fact that the logistic regressison for sklearn is better optimized than our implementation.

Deployment

We advise that SKlearns logistic regression implementation be used over ours. We made this decision due to the fact that SKlearns is much more optimized than our implementation since much of it is written in C, it has extensive reliability with various types of data, it has a lot of built in safety guards, and it has alot more tuning parameters than our implementation.

Another reason to use SKlearns over ours is the results from comparing both implementations, SKlearns was 6.356 times faster than ours in training and 2.269 times faster in testing. Even though the differences in accuracy is large the differences in times are enough to go with SKlearns over ours. The reason why timing is so important is because the faster the implementation is, the quicker it is to update a model therefore it is cheaper to maintain and cheaper to scale.

Exceptional Work

To accomplish calculating the optimal gradient in one iteration, we used a technique whose update is the inverse second derivative of the objective function multiplied by the first derivative. For a multi variable function, the second derivative is calculated with the Hessian Matrix. As shown in the figure, the hessian matrix can be calculated by taking the second derivative with respect to each variable combination. The figure shows the calculations in both matrix and index form.

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot \underbrace{\mathbf{H}[l(\mathbf{w})]^{-1}}_{\text{inverse Hessian}} \cdot \underbrace{\nabla l(\mathbf{w})}_{\text{gradient}}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot \underbrace{\left[\mathbf{X}^T \cdot \text{diag}\left[g(\mathbf{X}\cdot\mathbf{w})(1-g(\mathbf{X}\cdot\mathbf{w}))\right] \cdot \mathbf{X} - 2C\right]^{-1}}_{\text{inverse Hessian}} \cdot \underbrace{\mathbf{X} * y_{diff}}_{\text{gradient}}$$

All of our code is in the logistic regression class.

Hessian of MSE

Matrix. $\frac{\partial}{\partial w}(Y-Xw)'(Y-Xw) = 2X'(Y-Xw)$ — 1st deriv.

Indices $\frac{\partial}{\partial w_j}\left(\sum_{i=1}^{N} y_i - \sum_{j=1}^{D} x_{ij}w_j)\right)^2 = -2\sum_{i=1}^{N}\left(y_i - \sum_{j=1}^{D} x_{ij}w_j\right)x_{ij}$ — 1st deriv

Matrix $\frac{\partial}{\partial w \partial w'}(Y-Xw)'(Y-Xw) = \frac{\partial}{\partial w'} 2X'(Y-Xw) = 2X'X$ — 2nd deriv

Indices $\frac{\partial}{\partial w_j^2}\left(\sum_{i=1}^{N}(y_i - \sum_{j=1}^{D} x_{ij}w_j)\right)^2 = \frac{\partial}{\partial w_j}\left(-2\sum_{i=1}^{N}(y_i - \sum_{j=1}^{D} x_{ij}w_j)x_{ij}\right) = 2\sum_{i=1}^{N} x_{ij}^2$ — 2nd deriv

diagonal $\frac{\partial}{\partial w_j \partial w_k}\left(\sum_{i=1}^{N}(y_i - \sum_{j=1}^{D} x_{ij}w_j)\right)^2 = \frac{\partial}{\partial w_k}\left(-2\sum_{i=1}^{N}(y_i - \sum_{j=1}^{D} x_{ij}w_j)x_{ij}\right) = 2\sum_{i=1}^{N} x_{ij}x_{ik}$

Hessian: $2\sum_{i=1}^{N} x_{ij}^2$

$2\sum_{i=1}^{N} x_{ij}x_{ik}$

$$H(w_k) = \begin{bmatrix} 2x_{ij}x_{ik} & 2x^2_{i+1 j} & \cdots \\ 2x^2_{i j+1} & 2x_{i+1, j+1}x_{i+1, k+1} & \\ \vdots & & \ddots \end{bmatrix}$$

Run with train data

In [108]:
```
lr = LogisticRegression(optimization='one_hessian',eta=1e-06, regularization=
'ridge', iterations=1)
%time lr.fit(X_train, y_train)
pred = lr.predict(X_train)
encode = lambda x: lr.class_encodings_[x]
y_train_encode = np.array(list(map(encode, y_train)))
train_mse = accuracy_score(y_train_encode, pred)
print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".format
(train_mse,e,"one_hessian","ridge"))
```

```
Wall time: 1.36 s
Training MSE: 0.6784120633943683, Eta: 0.1, optimization: one_hessian, regula
rization: ridge
```

Run with test data

In [109]:
```
lr = LogisticRegression(optimization='one_hessian',eta=1e-06, regularization=
'ridge', iterations=1)
%time lr.fit(X_test, y_test)
pred = lr.predict(X_test)
encode = lambda x: lr.class_encodings_[x]
y_test_encode = np.array(list(map(encode, y_test)))
train_mse = accuracy_score(y_test_encode, pred)
print("Training MSE: {}, Eta: {}, optimization: {}, regularization: {}".format
(train_mse,e,"one_hessian","ridge"))
```

```
Wall time: 162 ms
Training MSE: 0.7132307692307692, Eta: 0.1, optimization: one_hessian, regula
rization: ridge
```

In [ ]:
```

```