1 Honor Code

Declare and sign the following statement:

"I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

Signature : Christian Aviña

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

2 Logistic Regression with Newton's Method

Given examples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and associated labels $y_1, y_2, \ldots, y_n \in \{0, 1\}$, the cost function for *unregularized* logistic regression is

$$J(\mathbf{w}) \triangleq -\sum_{i=1}^{n} \left(y_i \ln s_i + (1 - y_i) \ln(1 - s_i) \right)$$

where $s_i \triangleq s(\mathbf{x}_i \cdot \mathbf{w})$, $\mathbf{w} \in \mathbb{R}^d$ is a weight vector, and $s(\gamma) \triangleq 1/(1 + e^{-\gamma})$ is the logistic function.

Define the $n \times d$ design matrix X (whose i^{th} row is $\mathbf{x}_i^\top$), the label n-vector $\mathbf{y} \triangleq [y_1 \quad \ldots \quad y_n]^\top$, and $\mathbf{s} \triangleq [s_1 \quad \ldots \quad s_n]^\top$. For an n-vector $\mathbf{a}$, let $\ln \mathbf{a} \triangleq [\ln a_1 \quad \ldots \quad \ln a_n]^\top$. The cost function can be rewritten in vector form as

$$J(\mathbf{w}) = -\mathbf{y} \cdot \ln \mathbf{s} - (\mathbf{1} - \mathbf{y}) \cdot \ln (\mathbf{1} - \mathbf{s}).$$

Further, recall that for a real symmetric matrix $A \in \mathbb{R}^{d \times d}$, there exist U and Λ such that $A = U \Lambda U^\top$ is the eigendecomposition of A. Here Λ is a diagonal matrix with entries $\{\lambda_1, ..., \lambda_d\}$. An alternative notation is $\Lambda = diag(\lambda_i)$, where $diag()$ takes as input the list of diagonal entries, and constructs the corresponding diagonal matrix. This notation is widely used in libraries like `numpy`, and is useful for simplifying some of the expressions when written in matrix-vector form. For example, we can write $\mathbf{s} = diag(s_i)\,\mathbf{1}$.

Hint: See page two for notational conventions used here.

Hint: Recall matrix calculus identities. The elements in **bold** indicate vectors.

$$\nabla_{\mathbf{x}} \alpha \mathbf{y} = \big(\nabla_{\mathbf{x}} \alpha\big)\mathbf{y}^\top + \alpha \nabla_{\mathbf{x}} \mathbf{y} \qquad \nabla_{\mathbf{x}} \big(\mathbf{y} \cdot \mathbf{z}\big) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\mathbf{z} + \big(\nabla_{\mathbf{x}} \mathbf{z}\big)\mathbf{y};$$
$$\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{y}) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\big(\nabla_{\mathbf{y}} \mathbf{f}(\mathbf{y})\big); \qquad \nabla_{\mathbf{x}} g(\mathbf{y}) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\big(\nabla_{\mathbf{y}} g(\mathbf{y})\big);$$

and $\nabla_{\mathbf{x}} C\mathbf{y}(\mathbf{x}) = \big(\nabla_{\mathbf{x}} \mathbf{y}(\mathbf{x})\big) C^\top$, *where C is a constant matrix.*

1 Derive the gradient $\nabla_{\mathbf{w}} J(\mathbf{w})$ of cost $J(\mathbf{w})$ as a matrix-vector expression. Also derive *all intermediate derivatives* in matrix-vector form. Do NOT specify them (**including the intermediates**) in terms of their individual components (e.g. w_i for vector $\mathbf{w}$). **Solution:**

$s = \frac{1}{1+e^{(x \cdot w)}}$

We know that $s' = s(\gamma)(1 - s(\gamma))$ so,

$\nabla_w J(\mathbf{w}) = \frac{\partial f}{\partial w}[-y\ ln(s) + (1-y)\ ln(1-s)]$

$= -y(s) + (1-s)\frac{1}{s} \cdot x + (1-y)(-s(1-s)\frac{1}{1-s})$

$= -y(1-s)\mathbf{x} - (1-y)s\mathbf{x}$

$= (y - ys - s + ys)\mathbf{x}$

$= (\vec{y} - \vec{s})\vec{x}$

$\therefore \nabla w J(w) = x^T \cdot (y - s)$

2 Derive the Hessian $\nabla^2_{\mathbf{w}} J(\mathbf{w})$ for the cost function $J(\mathbf{w})$ as a matrix-vector expression.

Solution:

In vector form, $\nabla^J_w(w) = (\vec{y} - \vec{s})\vec{x}$

$\nabla^2_w J(w) = \frac{\partial f}{\partial w}(\vec{y} - \vec{s})\vec{x}$

$H = [-s(1-s)\vec{x}]\vec{x}$

$H = -x^T s(I-s)x$

3 Write the matrix-vector update law for one iteration of Newton's method, substituting the gradient and Hessian of $J(\mathbf{w})$.

Solution:

$(\nabla_w^2 J(w))e = -\nabla_w J(w)$

$e = -\frac{\nabla_w J(w)}{\nabla_w^2 J(w)}$

Update rule:

$w \leftarrow w + e$

$\therefore w \leftarrow w - \frac{x^T(\vec{y} - \vec{s})}{x^T s(I-s)x}$

4 You are given four examples $\mathbf{x}_1 = [0.2 \quad 3.1]^\top, \mathbf{x}_2 = [1.0 \quad 3.0]^\top, \mathbf{x}_3 = [-0.2 \quad 1.2]^\top, \mathbf{x}_4 = [1.0 \quad 1.1]^\top$ with labels $y_1 = 1, y_2 = 1, y_3 = 0, y_4 = 0$. These points cannot be separated by a line passing through origin. Hence, as described in lecture, append a 1 to each $\mathbf{x}_{i\in[4]}$ and use a weight vector $\mathbf{w} \in \mathbb{R}^3$ whose last component is the bias term (called α in lecture). Begin with initial weight $w^{(0)} = \begin{bmatrix}-1 & 1 & 0\end{bmatrix}^\top$. For the following, state only the final answer with four digits after the decimal point. You may use a calculator or write a program to solve for these, but do NOT submit any code for this part.

(a) State the value of $\mathbf{s}^{(0)}$ (the initial value of $\mathbf{s}$).

Solution: [0.9478, 0.8808, 0.8022, 0.5250]

(b) State the value of $\mathbf{w}^{(1)}$ (the value of $\mathbf{w}$ after 1 iteration).

Solution: [-7.3236, -2.5312, 5.4430]

(c) State the value of $\mathbf{s}^{(1)}$ (the value of $\mathbf{s}$ after 1 iteration).

Solution: [0.9802, 0.9526, 0.9168, 0.7503]

(d) State the value of $\mathbf{w}^{(2)}$ (the value of $\mathbf{w}$ after 2 iterations).

Solution: [-18.2168, -6.6412, 37.2602]

3 Wine Classification with Logistic Regression

The wine dataset `data.mat` consists of 6,000 sample points, each having 12 features. The description of these features is provided in `data.mat`. The dataset includes a training set of 5,000 sample points and a test set of 1,000 sample points. Your classifier needs to predict whether a wine is white (class label 0) or red (class label 1).

Begin by normalizing the data with each feature's mean and standard deviation. You should use training data statistics to normalize both training and validation/test data. Then add a fictitious dimension. Whenever required, it is recommended that you tune hyperparameter values with cross-validation.

Please set a random seed whenever needed and **report it**.

Use of automatic logistic regression libraries/packages is prohibited for this question. If you are coding in python, it is better to use `scipy.special.expit` for evaluating logistic functions as its code is numerically stable, and doesn't produce `NaN` or `MathOverflow` exceptions.

1 *Batch Gradient Descent Update.* State the batch gradient descent update law for logistic regression **with ℓ_2 regularization**. As this is a "batch" algorithm, each iteration should use *every training example*. You don't have to show your derivation. You may reuse results from your solution to question 2.1.

Solution:

Batch gradient descent update rule:

$w \leftarrow w + \epsilon x^T (y - s(x \cdot w))$

L2 Regularization derivative with respect to w:

$\frac{d}{dw} \frac{1}{2} \lambda ||w||^2$

$= \lambda w$

Batch gradient descent update rule with L2 regularization:

$w \leftarrow w + \epsilon x^T (y - s(x \cdot w) + \lambda w)$

2 *Batch Gradient Descent Code.* Implement your batch gradient descent algorithm for logistic regression and include your code here. Choose reasonable values for the regularization parameter and step size (learning rate), specify your chosen values in the write-up, and train your model from question 3.1. Shuffle and split your data into training/validation sets and mention the random seed used in the write-up. Plot the value of the cost function versus the number of iterations spent in training.

Solution: Seed used: np.random.seed(42)

```
"""
Performs Batch Gradient Descent Across Different Ranges to find optimal W with lowest cos
Makes predictions to kaggle using optimal w found
    """
sizes = [10, 100, 1000, 3000, 4000]

# add bias term to data
x_train_with_bias = np.hstack([np.ones((_X_train.shape[0], 1)), _X_train])
x_valid_with_bias = np.hstack([np.ones((_X_valid.shape[0], 1)), _X_valid])

logistic = LogisticRegression()
logistic.setX(x_train_with_bias)
logistic.setY(_Y_train)
logistic.setXValid(x_valid_with_bias)
logistic.setYValid(_Y_valid)
logistic.setWeights()
logistic.epsilon = 1e-1
logistic.Lambda = 1/3

currMinCost = logistic.LogisticRegressionCost()
optimalWeights = logistic.weights
overallCosts = []
costs = []

for n in sizes:
    costs = []
    for i in range(n):
        # Update weight vector
        # Calculate risk with new weight vector
        cost = logistic.LogisticRegressionCost()
        logistic.weights = logistic.batchGradientDescent()
        # if the risk decreased save the risk value and the optimal weights vector
        if cost < currMinCost:
            currMinCost = cost
            logistic.batch_optimal_w = logistic.weights.copy()
            logistic.mincost = cost
            # append the risk after n iterations to list of costs
        costs.append(currMinCost)
    overallCosts.append(min(costs))
print("Optimal Weights: ", logistic.batch_optimal_w)
print("Epsilon:", logistic.epsilon)
```

```
print("Lambda:", logistic.Lambda)
print("Minimum cost: ", round(logistic.mincost, 4))

fig = px.line(
    pd.DataFrame({"sizes" : sizes, "costs" : overallCosts}),
    x='sizes',
    y='costs',
    title='Size vs. Costs')
fig.update_layout(
    xaxis_title="Sizes",
    yaxis_title="Costs",
)
fig.show()
Output:
Optimal Weights:
[-1.89597506e+16  7.06804131e+15  7.24205590e+15 -2.10660254e+15
 -6.16628254e+15  4.28861956e+15 -1.72776184e+15 -1.25235091e+16
  9.14238416e+15  4.04858685e+15  5.57527696e+15  1.06873600e+15
  4.29767784e+15]
Epsilon: 0.1
Lambda: 0.3333333333333333
Minimum cost:  0.1224
```

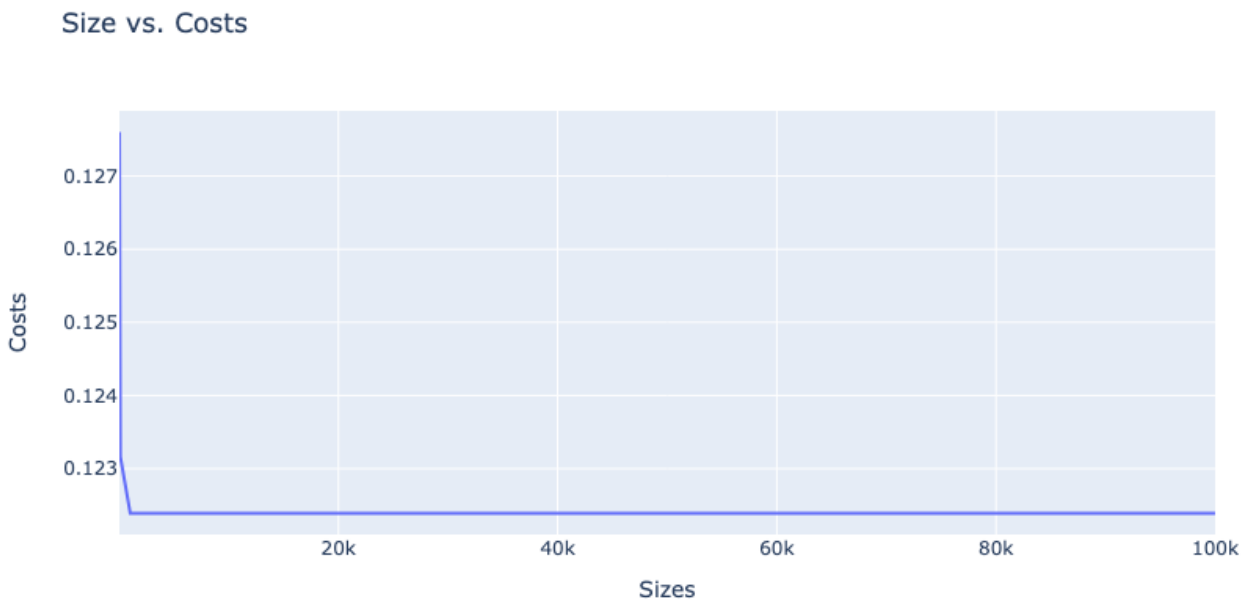

Figure 1: Iteration size vs. Costs

3 *Stochastic Gradient Descent (SGD) Update.* State the SGD update law for logistic regression with ℓ_2 regularization. Since this is not a "batch" algorithm anymore, each iteration uses *just one* training example. You don't have to show your derivation.

Solution:

$w \leftarrow w + \epsilon(y - s(x_i \cdot w)\ x_i + \lambda w)$

4 *Stochastic Gradient Descent Code.* Implement your stochastic gradient descent algorithm for logistic regression and include your code here. Choose a suitable value for the step size (learning rate), specify your chosen value in the write-up, and run your SGD algorithm from question 3.3. Shuffle and split your data into training/validation sets and mention the random seed used in the write-up. Plot the value of the cost function versus the number of iterations spent in training.

Compare your plot here with that of question 3.2. Which method converges more quickly? Briefly describe what you observe.

Solution:

```
#  Shuffle and split normalized non pca data with no bias term
data, labels = shuffleMe(X_train_normalized, y_train)

# split shuffled data into validation and training sets
stochastic_validation_data = data[-80:]
stochastic_validation_labels = labels[-80:]
print(stochastic_validation_data.shape)
print(stochastic_validation_labels.shape)
stochastic_training_data = data[:-80]
stochastic_training_labels = labels[:-80]
print(stochastic_training_data.shape)
print(stochastic_training_labels.shape)

# Declare logistic regression object
stochastic_logistic_regression = LogisticRegression()
stochastic_logistic_regression.setX(stochastic_training_data)
stochastic_logistic_regression.setY(stochastic_training_labels)
stochastic_logistic_regression.setXValid(stochastic_validation_data)
stochastic_logistic_regression.setYValid(stochastic_validation_labels)
stochastic_logistic_regression.setWeights()
stochastic_logistic_regression.epsilon = 1e-5
stochastic_logistic_regression.Lambda = 1/5

print('initial weights: ', stochastic_logistic_regression.weights)

stochastic_logistic_regression.stochastic_cost = stochastic_logistic_regression.LogisticR


stochastic_costs = []
stochastic_accuracy = -1

for i in range(stochastic_logistic_regression.X.shape[0]):
    stochastic_logistic_regression.weights = stochastic_logistic_regression.StochasticGra
    temp_cost = stochastic_logistic_regression.LogisticRegressionCost()
    temp_accuracy = stochastic_logistic_regression.getAccuracy()

    if temp_accuracy > stochastic_accuracy:
        stochastic_accuracy = temp_accuracy

    if temp_cost < stochastic_logistic_regression.stochastic_cost:
```

```
            stochastic_logistic_regression.stochastic_cost = temp_cost

        stochastic_costs.append(temp_cost)
    print("Minimum cost with stochastic gradient descent:", round(stochastic_logistic_regressi

    t = range(4000)
    print(len(stochastic_costs))
    fig_3_4 = px.scatter(pd.DataFrame({'Index' :t
                                       ,'Cost' : stochastic_costs})
                                       , x ='Index'
                                       , y = 'Cost')
    fig_3_4.update_layout(
        xaxis_title = 'Training Point No.',
        yaxis_title = 'Cost'
    )

    fig_3_4.show()
```

Output:

```
Minimum cost with stochastic gradient descent: 0.2323
```

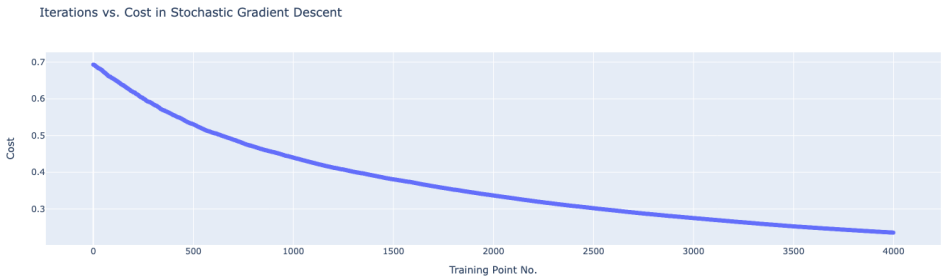

Figure 2: 3.4 Stochastic Gradient Descent

5 Instead of using a constant step size (learning rate) in SGD, you could use a step size that slowly shrinks from iteration to iteration. Run your SGD algorithm from question 3.3 with a step size $\epsilon_t = \delta/t$ where t is the iteration number and δ is a hyperparameter you select empirically. Mention the value of δ chosen. Plot the value of cost function versus the number of iterations spent in training.

How does this compare to the convergence of your previous SGD code?

Solution:

Cost converges faster but to a slightly higher cost than batch gradient descent.

Min cost with stochastic gradient descent: 0.2417

Min cost with batch gradient descent: 0.1224

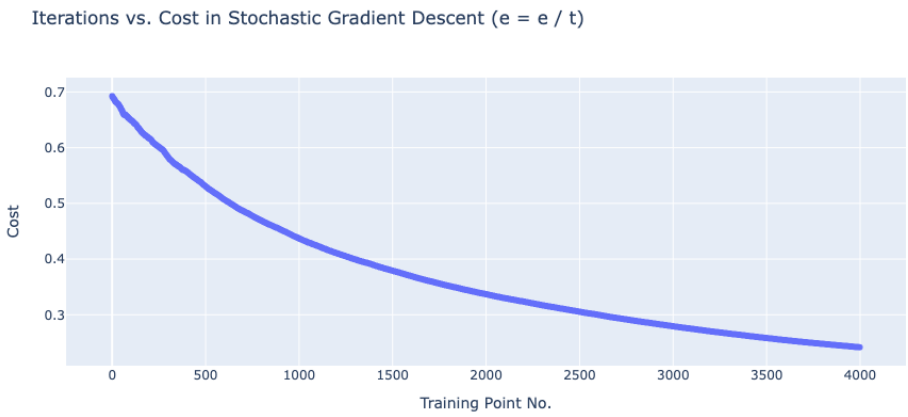

Figure 3: Enter Caption

6 *Kaggle.* Train your *best* classifier on the entire training set and submit your prediction on the test sample points to Kaggle. As always for Kaggle competitions, you are welcome to add or remove features, tweak the algorithm, and do pretty much anything you want to improve your Kaggle leaderboard performance **except** that you may not replace or augment logistic regression with a wholly different learning algorithm. Your code should output the predicted labels in a CSV file.

Report your Kaggle username and your best score, and briefly describe what your best classifier does to achieve that score.

Solution:

Kaggle username: christianreyesavina

Kaggle Best Score: 0.99

My best classifier is quite simple.

First I split the training data into a validation and training set. I performed PCA on both sets, and projected them onto their respective eigenvectors. I was able to capture 82.57 of the variance in the original data and projected the data onto its pca eigenvectors. This reduced the 12 dimensional data to 6 dimensions. The reason I was able to achieve .99 accuracy is because I implemented a PCA transformation on both the training set and the test set. Therefore, the logistic model was fit with highly distilled data that reduced training time, captured variance, and made highly accurate predictions on the test set.

```
# Pca
print('Training Data PCA')
training_pca = PCA()
x_training_pca = training_pca.pca(X_train_norm_shuffled)
print(x_training_pca.shape)
# pca on the true test set

# project test data onto training eigenvectors
x_test_pca = X_test_normalized_no_bias @ training_pca.eigenvectors

# entire training set in pca form
_x_train_PCA = x_training_pca[:_80]

_y_train_PCA = y_train_shuffled[:_80]
# Set aside 20% of data training data for validation"
_x_valid_PCA = x_training_pca[_80:]
_y_valid_PCA = y_train_shuffled[_80:]

logisticPCA = LogisticRegression()
# Model is fit with only 100_000 iterations
x_train_pca_with_bias = np.hstack([np.ones((_x_train_PCA.shape[0], 1)), _x_train_PCA])
x_valid_pca_with_bias = np.hstack([np.ones((_x_valid_PCA.shape[0], 1)), _x_valid_PCA])

logisticPCA.fit(x_train_pca_with_bias,
        _y_train_PCA,
        x_valid_pca_with_bias,
        _y_valid_PCA,
        epsilon= 1e-3,
        lam=1/9)
```

```
x_test_pca_with_bias = np.hstack([np.ones((x_test_pca.shape[0], 1)), x_test_pca])

logisticKagglePredictions = logisticPCA.predict(x_test_pca_with_bias)
resultsToCsv(logisticKagglePredictions)
pd.Series(logisticKagglePredictions).value_counts()
print('Accuracy on validation set: ', logisticPCA.getAccuracy())
print('Kaggle Score: .99')
Output:

0    738
1    262
dtype: int64
Accuracy on validation set:  0.985
Kaggle Accuracy: .99
```

4 A Bayesian Interpretation of Lasso

Suppose you are aware that the labels $y_{i\in[n]}$ corresponding to sample points $\mathbf{x}_{i\in[n]} \in \mathbb{R}^d$ follow the density law

$$f(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(y_i - \mathbf{w}\cdot\mathbf{x}_i)^2/(2\sigma^2)}$$

where $\sigma > 0$ is a known constant and $\mathbf{w} \in \mathbb{R}^d$ is a random parameter. Suppose further that experts have told you that

- each component of $\mathbf{w}$ is independent of the others, and
- each component of $\mathbf{w}$ has the Laplace distribution with location 0 and scale being a known constant b. That is, each component $\mathbf{w}_i$ obeys the density law $f(\mathbf{w}_i) = e^{-|\mathbf{w}_i|/b}/(2b)$.

Assume the outputs $y_{i\in[n]}$ are independent from each other.

Your goal is to find the choice of parameter $\mathbf{w}$ that is *most likely* given the input-output examples $(\mathbf{x}_i, y_i)_{i\in[n]}$. This method of estimating parameters is called *maximum a posteriori* (MAP); Latin for *"maximum [odds] from what follows."*

1. Derive the *posterior* probability density law $f(\mathbf{w}|(\mathbf{x}_i, y_i)_{i\in[n]})$ for $\mathbf{w}$ *up to a proportionality constant*

   by applying Bayes' Theorem and substituting for the densities $f(y_i|\mathbf{x}_i, \mathbf{w})$ and $f(\mathbf{w})$. Don't try to derive an exact expression for $f(\mathbf{w}|(\mathbf{x}_i, y_i)_{i\in[n]})$, as the denominator is very involved and irrelevant to maximum likelihood estimation.

   Solution:

   The posterior probability $f(\mathrm{w—x}_i y_i) = \frac{f(y_i|x_i w)f(w)}{f(y_i|x_i)}$

   Where, $f(y_i|x_i) = \int f(y_i|x_i, w)f(w)f(w)dw$

   $f(w|x_i y_i)_{i\in[N]} = \frac{f(y_i|x_i,w)f(w)}{\int f(y_i|x_i,w)_{i\in[N]}}$

   $= \frac{\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x_i-y_i)^2}{2\sigma^2}} e^{\frac{-|w_i|}{b}}}{\int (y_i|x_i,w)_{i\in[N]}}$

2. Define the log-likelihood for MAP as $\ell(\mathbf{w}) \triangleq \ln f(\mathbf{w}|\mathbf{x}_{i\in[n]}, y_{i\in[n]})$. Show that maximizing the MAP log-likelihood over all choices of $\mathbf{w}$ is the same as minimizing $\sum_{i=1}^{n}(y_i - \mathbf{w}\cdot\mathbf{x}_i)^2 + \lambda\|\mathbf{w}\|_1$ where $\|\mathbf{w}\|_1 = \sum_{j=1}^{d}|w_j|$ and λ is a constant. Also give a formula for λ as a function of the distribution parameters.

Solution:

Maximizing MAP:

$$\ell(w) = ln\left(\frac{\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x_i-y_i)^2}{2\sigma^2}}e^{\frac{-|w_i|}{b}}}{\int(y_i|x_i,w)_{i\in[N]}}\right)$$

$$\nabla_w\ell(w) = \frac{\partial}{\partial w}ln\left(\frac{\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x_i-y_i)^2}{2\sigma^2}}e^{\frac{-|w_i|}{b}}}{\int(y_i|x_i,w)_{i\in[N]}}\right)$$

$$= ln(\frac{1}{\sigma\sqrt{2\pi}}) + ln(e^{\frac{-(x_i-y_i)^2}{2\sigma^2}}) + \ln(\frac{e^{-\frac{|w_i|}{b}}}{2b})$$

$$= \frac{\partial}{\partial w}\left[ln(\frac{1}{\sigma\sqrt{2\pi}}) - \frac{(x_i-yi)^2}{2\sigma^2} + ln(\frac{1}{2b}) - (\frac{w_i}{b})\right]$$

$$= \frac{d}{dw}\frac{|w_i|}{b} = \begin{cases}\frac{1}{b} \text{ if w} > 0\\ -\frac{1}{b} \text{ if w} < 0\end{cases}$$

5 ℓ_1-regularization, ℓ_2-regularization, and Sparsity

You are given a design matrix X (whose i^{th} row is sample point $\mathbf{x}_i^\top$) and an n-vector of labels $\mathbf{y} \triangleq [y_1 \quad \dots \quad y_n]^\top$. For simplicity, assume X is whitened, so $X^\top X = nI$. Do not add a fictitious dimension/bias term; for input $\mathbf{0}$, the output is always 0. Let $\mathbf{x}_{*i}$ denote the i^{th} column of X.

1 The ℓ_p-norm for $w \in \mathbb{R}^d$ is defined as $||w||_p = (\sum_{i=1}^d |w_i|^p)^{1/p}$, where $p > 0$. Plot the isocontours with $w \in \mathbb{R}^2$, for the following norms.
(a) $\ell_{0.5}$ (b) ℓ_1 (c) ℓ_2
Use of automatic libraries/packages for computing norms is prohibited for the question.

Solution:

$(a) \ell_{0.5} = 0.40$

$(b) \ell_1 = 2.0$

$(c) \ell_2 = 1.4142$

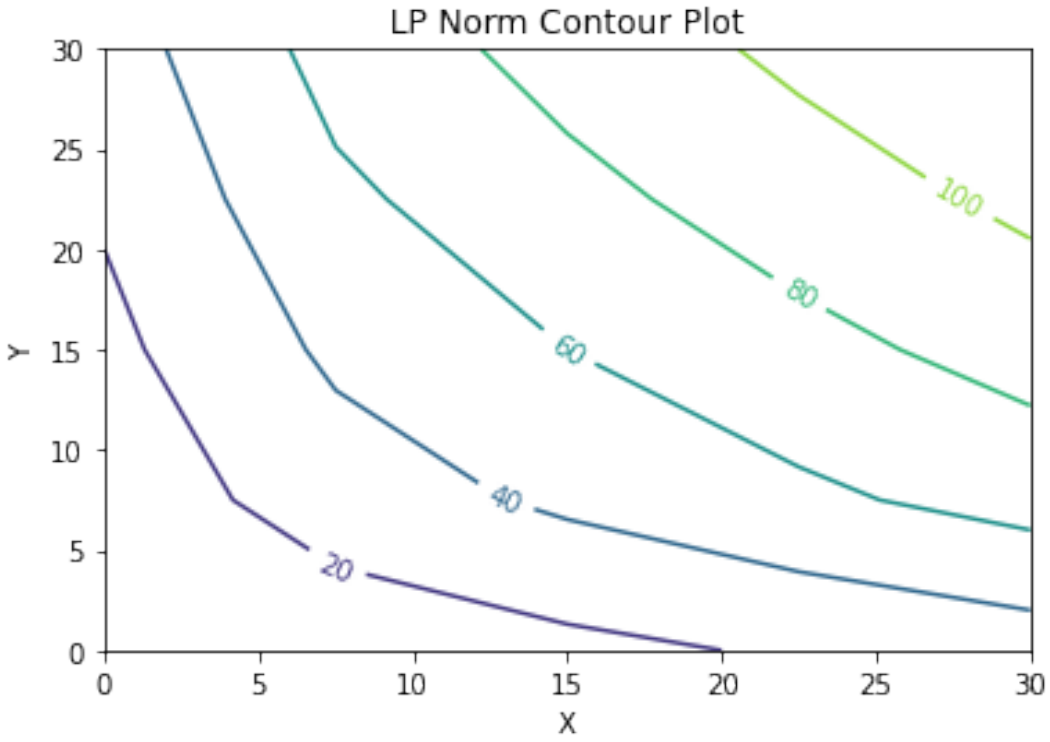

Figure 4: $\ell_{0.5}$

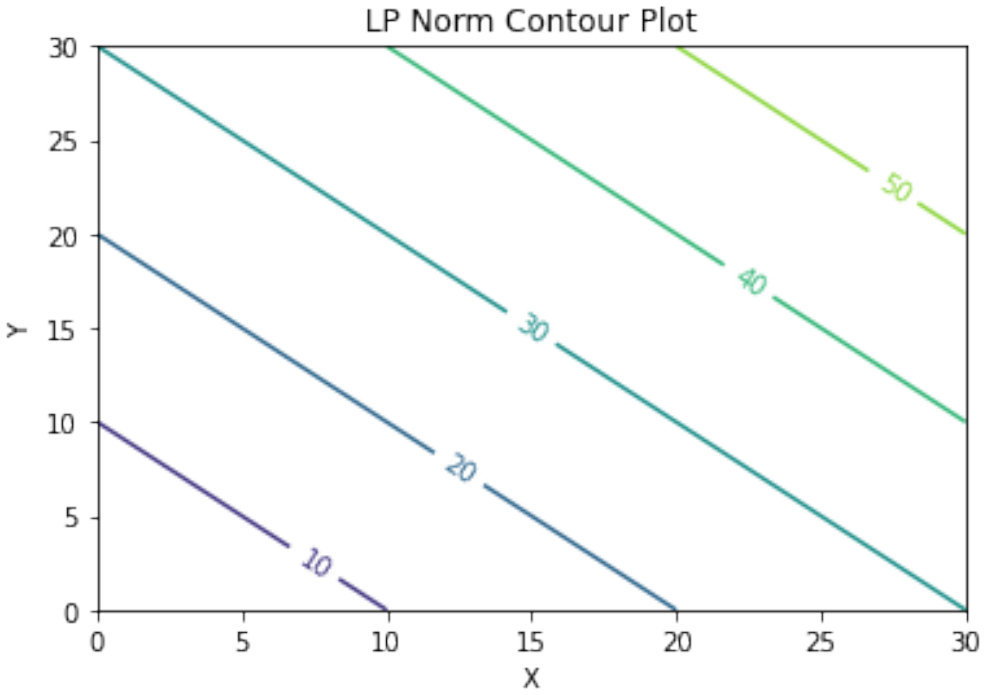

Figure 5: $\ell_{1.0}$

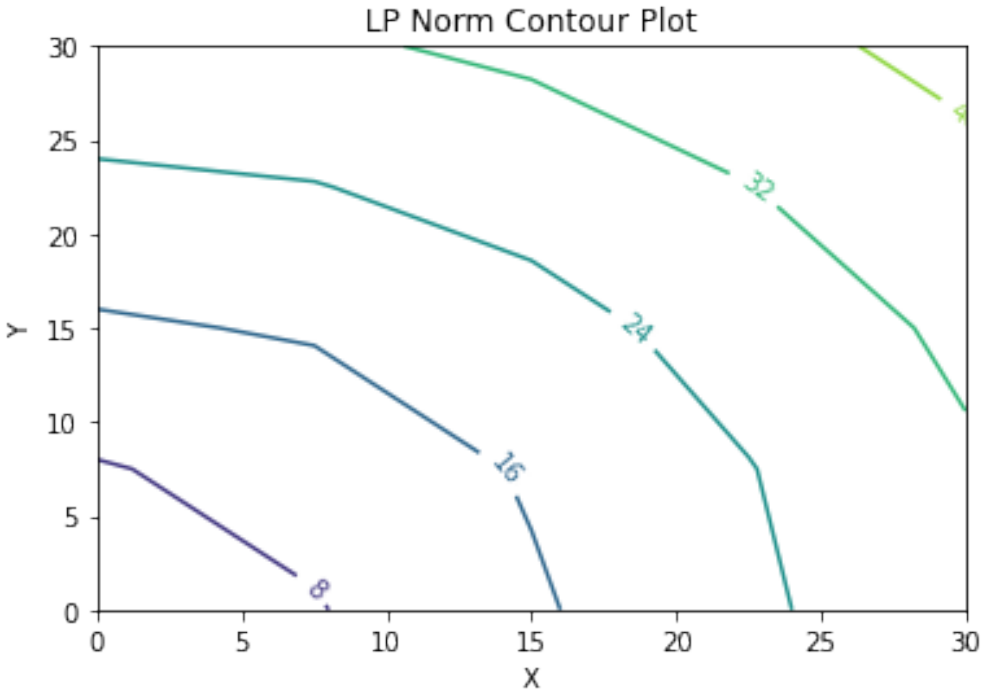

Figure 6: ℓ_2

2 Show that the cost function for ℓ_1-regularized least squares, $J_1(\mathbf{w}) \triangleq \|X\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|_1$ (where $\lambda > 0$), can be rewritten as $J_1(\mathbf{w}) = \|\mathbf{y}\|^2 + \sum_{i=1}^{d} f(\mathbf{x}_{*i}, \mathbf{w}_i)$ where $f(\cdot, \cdot)$ is a suitable function whose first argument is a vector and second argument is a scalar.

Solution:

For the entire dataset: $||Xw - y||^2 + \lambda||w||$

The norm $||w_i||_p = \sum_{j=1}^{d}(w_i^p)^{\frac{1}{p}}$

For a single datapoint:

The norm $||w_i||_p = \sum_{j=1}^{d}(x_i w_i^p)^{\frac{1}{p}}$

Therefore, the cost function can be written as:

$||y||^2 + \sum_{i=1}^{d} f(x_{i,w_i})$

Where $f(\cdot, \cdot) = \sum_{i=1}^{d}(x_i \cdot w_i^p)^{\frac{1}{p}}$

3 Using your solution to part 2, derive necessary and sufficient conditions for the i^{th} component of the optimizer $\mathbf{w}^*$ of $J_1(\cdot)$ to satisfy each of these three properties: $w_i^* > 0, w_i^* = 0$, and $w_i^* < 0$.

Solution:

The conditions for the i_{th} component of the optimizer of $J_1(\cdot)$

For $w_i^* > 0$ The ℓ_p term must be positive.

For $w_i^* = 0$ Not possible. ℓ_0 yields positive w_i

For $w_i^* < 0$ Not possible. $\ell_p > 0$

4 For the optimizer $\mathbf{w}^{\#}$ of the ℓ_2-regularized least squares cost function $J_2(\mathbf{w}) \triangleq \|X\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|^2$ (where $\lambda > 0$), derive a necessary and sufficient condition for $\mathbf{w}_i^{\#} = 0$, where $\mathbf{w}_i^{\#}$ is the ith component of $\mathbf{w}^{\#}$.

Solution:

$J_2(w) = ||Xw - y||_2^2 + \lambda||w||_2^2$

$= \sum_{i=1}^{d}(x_i \cdot w_i^p)^{\frac{1}{p}}$, where $p > 0$.

$\therefore$ there are no values that can make $w_i^{\#} = 0$.

5 A vector is called *sparse* if most of its components are 0. From your solution to part 3 and 4, which of $\mathbf{w}^*$ and $\mathbf{w}^\#$ is more likely to be sparse? Why?

Solution:

Neither w^* nor $w^\#$ are likely to be sparse because each component of w is found using $||w_i||_p = \sum_{i=d}^{d}(w_i^p)^{\frac{1}{p}}$ and since $p > 0$, then all elements are non-negative. Therefore, there will be no sparsity.

Code Appendix:

```
import os
import scipy.io
import matplotlib.pyplot as plt
from scipy.special import expit
import numpy as np
import copy
import pandas as pd
import plotly.express as px
from sklearn.metrics import accuracy_score
np.random.seed(42)

def shuffleMe(data, labels):
    """Function shuffles the input array and returns a shuffled array"""
    indices = np.random.permutation(len(data))
    shuffled_data = data[indices]
    shuffled_labels = labels[indices]
    return shuffled_data, shuffled_labels
def resultsToCsv(y_test):
    y_test = y_test.astype(int)
    df = pd.DataFrame({'Category': y_test})
    df.index += 1 # Ensures that the index starts at 1
    df.to_csv(f"/Users/christian/Desktop/CS189/Homeworks/hw4/wine_predictions.csv", index_lab
def NormalizeData(X):
    # NORMALIZE THE ENTIRE TRAINING
    X_Normalized = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
    return X_Normalized

class PCA:
    def __init__(self, variance=0.80):
        self.data = None
        self.data_transformed = None
        self.variance = variance  # Proportion of variance to retain
        self.eigenvalues = None
        self.eigenvectors = None
        self.covariance_matrix = None

    def pca(self, data):
        self.data = data
        self.covariance_matrix = np.cov(self.data, rowvar=False)
        self.eigenvalues, self.eigenvectors = np.linalg.eig(self.covariance_matrix)
        idx = np.argsort(self.eigenvalues)[::-1]
        self.eigenvalues = self.eigenvalues[idx]
        self.eigenvectors = self.eigenvectors[:, idx]

        total_variance = np.sum(self.eigenvalues)
        variance_ratios = self.eigenvalues / total_variance
        cumulative_variance = np.cumsum(variance_ratios)
        num_components = np.where(cumulative_variance >= self.variance)[0][0] + 1
```

```
        self.data_transformed = self.data @ self.eigenvectors[:, :num_components]
        self.eigenvectors = self.eigenvectors[:, :num_components]


        # Print information
        print("Eigenvalues:", self.eigenvalues[:num_components])
        print("Number of components to retain:", num_components)
        print("Cumulative variance explained by these components:", cumulative_variance[num_c

        return self.data_transformed

# Change working directory
os.chdir("/Users/christian/Desktop/CS189/Homeworks/hw4")
os.getcwd()
os.listdir()


# load mat file
data = scipy.io.loadmat('data.mat')
# Extract training data from mat file
X_train = data['X']
# Extract and flatten the training labels in the mat file
y_train = data['y'].flatten()
# Extract true test data from mat file
X_test = data['X_test']

# Normalize entire dataset
X_train_normalized = NormalizeData(X_train)

# shuffle the entire dataset
X_train_norm_shuffled, y_train_shuffled = shuffleMe(X_train_normalized, y_train)

# Normalize true test data
X_test_normalized_no_bias = (X_test - np.mean(X_train, axis=0)) / np.std(X_train, axis=0)

#  80% of the data
_80 = int(len(X_train_norm_shuffled) * .80)

# Non PCA data
_X_train = X_train_norm_shuffled[:_80]
_Y_train = y_train_shuffled[:_80]
_X_valid  = X_train_norm_shuffled[_80:]
_Y_valid = y_train_shuffled[_80:]


class LogisticRegression():
    def __init__(self):
        # shuffle the normalized training data and labels
```

```
        self.X = None
        self.Y = None
        self.X_valid = None
        self.Y_valid = None
        self.epsilon = None
        self.Lambda = None
        self.weights = None
        self.batch_optimal_w = None
        self.stochastic_optimal_w = None
        self.mincost = None
        self.stochastic_cost = None
        self.topAccuracy = None
    # setter functions
    def setX(self, X):
        self.X = X
    def setY(self, Y):
        self.Y = Y
    def setXValid(self, X_valid):
         self.X_valid = X_valid
    def setYValid(self, Y_valid):
        self.Y_valid = Y_valid
    def setWeights(self):
        """Sets weight vectors (no bias term)"""
        self.weights = np.zeros(self.X.shape[1])
    # getter functions
    def getWeights(self):
        return self.weights
    def getXValid(self):
        return self.X_valid
    def getXTrain(self):
        return self.X
    def getValidationLabels(self):
        return self.Y_valid
    def batchGradientDescent(self):
        s = expit(self.X @ self.weights)
        gradient = (self.X.T @ (self.Y - s)) + (self.Lambda * self.weights)
        # Im pretty sure the gradient is negative which is why i use a plus sign
        return (self.weights + (self.epsilon * gradient))
    def LogisticRegressionCost(self):
        predictions = expit(self.X @ self.weights)
        # Clip predictions to avoid log(0)
        predictions_clipped = np.clip(predictions, self.epsilon, 1 - self.epsilon)
        return -np.mean(self.Y * np.log(predictions_clipped) + (1 - self.Y) * np.log(1 - pred
    def getAccuracy(self):
        """Returns the accuracy on the validation set"""
        threshold = 0.5
        predictions = expit(self.X_valid @ self.weights)
        predictionClipped = np.clip(predictions, self.epsilon, 1 - self.epsilon)
        # Convert probabilities to binary class labels
```

```
        y_hat  = (predictionClipped >= threshold).astype(int)
        accuracy = accuracy_score(self.Y_valid, y_hat)
        return accuracy
    def StochasticGradientDescent(self, i):
        """Returns optimal w by training on every point (stochastic gradient descent)"""
        s = expit(self.X[i] @ self.weights)
        new_w =  self.weights + (1e-3 * ((self.Y[i] - s) * self.X[i] + (1/4 * self.weights)))
        return new_w
    def fit(self, X, Y, X_valid, Y_valid, epsilon, lam):
        """
        Code for Kaggle Predictions
        Takes training and validation data in PCA form with no bias term.
        Finds an optimal weight vector that optimizes accuracy on the validation set in PCA fo

        Parameters:
        X (np.array): Input training Data (in this case with no bias term because it's in PCA
        Y (np.array): Input labels for training data.
        X_valid (np.array) : Input validation data.
        Y_valid (np.array) : Input valudation data labels.
        """
        self.epsilon = epsilon
        self.Lambda = lam
        self.setX(X)
        self.setY(Y)
        self.setXValid(X_valid)
        self.setYValid(Y_valid)
        self.setWeights()
        self.mincost= -1

        cost = - 1
        topAcc = self.getAccuracy()
        #print('dot product of features and training labels')
        #print(trainingFeatures.T @ trainingLabels)
        for i in range(100_000):
            self.weights = self.batchGradientDescent()
            cost = self.LogisticRegressionCost()
            acc = self.getAccuracy()
            if acc > topAcc:
                topAcc = acc
                self.topAccuracy = acc
                self.optimalW = self.weights
                self.mincost = cost

    def predict(self, data):
        """Makes predictions on training, test, validation normalized test/validation sets"""
        #np.hstack([np.ones((data.shape[0], 1)), data])
        predictions = expit(data @ self.optimalW)
        predictionsClipped = np.clip(predictions, self.epsilon, 1 - self.epsilon)
        y_hat = (predictionsClipped >= 0.5).astype(int)
```

```
        return y_hat



"""
Performs Batch Gradient Descent Across Different Ranges to find optimal W with lowest cost usi
Makes predictions to kaggle using optimal w found
    """
sizes = [10, 100, 1000, 3000, 4000]

# add bias term to data
x_train_with_bias = np.hstack([np.ones((_X_train.shape[0], 1)), _X_train])
x_valid_with_bias = np.hstack([np.ones((_X_valid.shape[0], 1)), _X_valid])

logistic = LogisticRegression()
logistic.setX(x_train_with_bias)
logistic.setY(_Y_train)
logistic.setXValid(x_valid_with_bias)
logistic.setYValid(_Y_valid)
logistic.setWeights()
logistic.epsilon = 1e-1
logistic.Lambda = 1/3

currMinCost = logistic.LogisticRegressionCost()
optimalWeights = logistic.weights
overallCosts = []
costs = []

for n in sizes:
    costs = []
    for i in range(n):
        # Update weight vector
        # Calculate risk with new weight vector
        cost = logistic.LogisticRegressionCost()
        logistic.weights = logistic.batchGradientDescent()
        # if the risk decreased save the risk value and the optimal weights vector
        if cost < currMinCost:
            currMinCost = cost
            logistic.batch_optimal_w = logistic.weights.copy()
            logistic.mincost = cost
            # append the risk after n iterations to list of costs
        costs.append(currMinCost)
    overallCosts.append(min(costs))
print("Optimal Weights: ")
print(logistic.batch_optimal_w)
print("Epsilon:", logistic.epsilon)
print("Lambda:", logistic.Lambda)
print("Minimum cost: ", round(logistic.mincost, 4))
```

```
fig = px.line(
    pd.DataFrame({"sizes" : sizes, "costs" : overallCosts}),
    x='sizes',
    y='costs',
    title='Size vs. Costs')
fig.update_layout(
    xaxis_title="Sizes",
    yaxis_title="Costs",
)
fig.show()

#  Shuffle and split normalized non pca data with no bias term
data, labels = shuffleMe(X_train_normalized, y_train)

# split shuffled data into validation and training sets
stochastic_validation_data = data[-80:]
stochastic_validation_labels = labels[-80:]
stochastic_training_data = data[:-80]
stochastic_training_labels = labels[:-80]

# Declare logistic regression object
stochastic_logistic_regression = LogisticRegression()
stochastic_logistic_regression.setX(stochastic_training_data)
stochastic_logistic_regression.setY(stochastic_training_labels)
stochastic_logistic_regression.setXValid(stochastic_validation_data)
stochastic_logistic_regression.setYValid(stochastic_validation_labels)
stochastic_logistic_regression.setWeights()
stochastic_logistic_regression.epsilon = 1e-5
stochastic_logistic_regression.Lambda = 1/5

stochastic_logistic_regression.stochastic_cost = stochastic_logistic_regression.LogisticRegres

stochastic_costs = []
stochastic_accuracy = -1

for i in range(stochastic_logistic_regression.X.shape[0]):
    stochastic_logistic_regression.weights = stochastic_logistic_regression.StochasticGradient
    stochastic_logistic_regression.epsilon = stochastic_logistic_regression.epsilon
    temp_cost = stochastic_logistic_regression.LogisticRegressionCost()
    temp_accuracy = stochastic_logistic_regression.getAccuracy()

    if temp_accuracy > stochastic_accuracy:
        stochastic_accuracy = temp_accuracy

    if temp_cost < stochastic_logistic_regression.stochastic_cost:
        stochastic_logistic_regression.stochastic_cost = temp_cost

    stochastic_costs.append(temp_cost)
print("Minimum cost with stochastic gradient descent:", round(stochastic_logistic_regression.
```

```
t = range(4000)

fig_3_4 = px.scatter(pd.DataFrame({'Index' :t
                                   ,'Cost' : stochastic_costs})
                                   , x ='Index'
                                   , y = 'Cost'
                                   , title='Iterations vs. Cost in Stochastic Gradient Descen
fig_3_4.update_layout(
    xaxis_title = 'Training Point No.',
    yaxis_title = 'Cost'
)

fig_3_4.show()


# Pca
print('Training Data PCA')
training_pca = PCA()
x_training_pca = training_pca.pca(X_train_norm_shuffled)
print(x_training_pca.shape)
# pca on the true test set

# project test data onto training eigenvectors
x_test_pca = X_test_normalized_no_bias @ training_pca.eigenvectors

# entire training set in pca form
_x_train_PCA = x_training_pca[:_80]

_y_train_PCA = y_train_shuffled[:_80]
# Set aside 20% of data training data for validation"
_x_valid_PCA = x_training_pca[_80:]
_y_valid_PCA = y_train_shuffled[_80:]

logisticPCA = LogisticRegression()
# Model is fit with only 100_000 iterations
x_train_pca_with_bias = np.hstack([np.ones((_x_train_PCA.shape[0], 1)), _x_train_PCA])
x_valid_pca_with_bias = np.hstack([np.ones((_x_valid_PCA.shape[0], 1)), _x_valid_PCA])

logisticPCA.fit(x_train_pca_with_bias, _y_train_PCA, x_valid_pca_with_bias, _y_valid_PCA, eps

x_test_pca_with_bias = np.hstack([np.ones((x_test_pca.shape[0], 1)), x_test_pca])

logisticKagglePredictions = logisticPCA.predict(x_test_pca_with_bias)
resultsToCsv(logisticKagglePredictions)

print(pd.Series(logisticKagglePredictions).value_counts())
print('Accuracy on validation set: ', logisticPCA.getAccuracy())
print('Kaggle Score: .99')
```

```
def NormContourPlot(norm):
    # Define the grid
    x_range = np.linspace(0, 30, 5)
    y_range = np.linspace(0, 30, 5)
    X, Y = np.meshgrid(x_range, y_range)

    Z = np.zeros(X.shape)
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            x = np.array([X[i, j], Y[i, j]])
            Z[i, j] = (sum(x**norm)**(1/norm))

    # Plot the contour plot
    contours = plt.contour(X, Y, Z, levels=5)
    plt.clabel(contours, inline=True, fontsize=10)
    plt.title('LP Norm Contour Plot')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.autoscale(enable=True, tight=True)
    plt.show()

# Assume X_train is your training data
data_centered = X_train - np.mean(X_train, axis=0)

# Compute the covariance matrix
covariance_matrix = np.cov(data_centered, rowvar=False)

# Perform the SVD
U, S, Vt = np.linalg.svd(covariance_matrix)

# Create the D matrix which is the inverse square root of the eigenvalues matrix
D = np.diag(1.0 / np.sqrt(S))

# Whiten the data
data_whitened = data_centered @ U @ D @ Vt

norms = [0.5, 1, 2]
lps = []

for p in norms:
    temp_w = np.ones(2)
    lps.append(sum(temp_w**p) ** (1/p))


print(lps)

NormContourPlot(norms[0])
NormContourPlot(norms[1])
```

```
NormContourPlot(norms[2])
```