Eric Kong

Data 102
October 8, 2019

Homework 3

Problem 1

1. $P(\beta|X) = \frac{P(X|\beta)P(\beta)}{P(X)}$ where $P(X)$ is part of the proportionality constant.
   $P(\beta) = N(0, \sigma_\beta^2)$
   $P(X|\beta) = lik(\beta) =$

2. $argmin_\beta \; ||X\beta - y||_2^2 + \lambda||\beta||_2^2 = \frac{d}{d\beta}||X\beta - y||_2^2 + \lambda||\beta||_2^2 =$
   $2X^T(X\beta - y) + 2\lambda\beta = 0$
   $X^T X\beta - X^T y + \lambda\beta = 0$
   $X^T y = \beta(X^T X + \lambda)$
   $\beta = X^T y(X^T X + \lambda)^{-1}$

3. $\lambda = -X^T(X\beta - y)\beta^{-1}$

Problem 2

In [26]:
```
def regression_predict(X, beta):
    """Given a linear model (aka a vector) and a feature matrix
    predict the output vector.

    Parameters
    ----------
    X : numpy array of shape nxd
        The feature matrix where each row corresponds to a single
        feature vector.
    beta : numpy array of shape d
        The linear model.

    Returns
    -------
    y : numpy array of shape n
        The predicted output vector.
    """
    # TODO: Fill in (Q2a)
    return X @ beta

def regression_least_squares(X, true_y, lambda_value):
    """Compute the optimal linear model that minimizes the regularized squared loss.

    Parameters
    ----------
    X : numpy array of shape nxd
        The feature matrix where each row corresponds to a single
        feature vector.
    true_y : numpy array of shape n
        The true output vector.
    lambda_value : float
        A non-negative regularization term.

    Returns
    -------
    beta : numpy array of shape d
        The optimal linear model.
    """
    # TODO: Fill in (Q2a)
    beta = (X.T @ true_y) @ np.linalg.inv(np.dot(X.T, X) + lambda_value)
    return beta
```

1.

2.

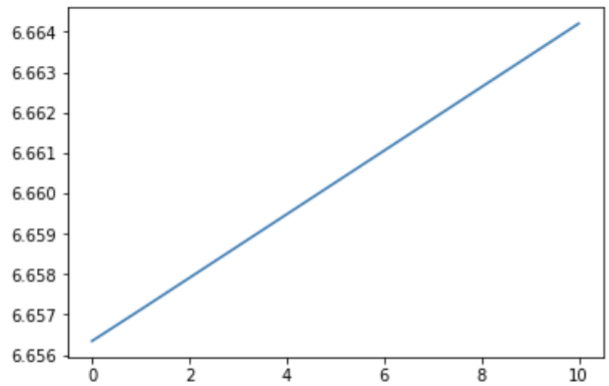
In [6]:
```
def sigmoid(x):
    """The sigmoid function."""
    return 1 / (1 + np.exp(-x))

def logistic_predict(X, beta):
    """Given a linear model (aka a vector) and a feature matrix
    predict the probability of the output label being 1 using logistic
    regression.

    Parameters
    ----------
    X : numpy array of shape nxd
        The feature matrix where each row corresponds to a single
        feature vector.
    beta : numpy array of shape d
        The linear model.

    Returns
    -------
    y : numpy array of shape n
        The predicted output vector.
    """
    # TODO: Fill in (Q2b)
    y = sigmoid(X @ beta)
    return y

def logistic_cross_entropy_loss(X, beta, true_y):
    """Output the cross entropy loss of a given logistic model.

    Parameters
    ----------
    X : numpy array of shape nxd
        The feature matrix where each row corresponds to a single
        feature vector.
    beta : numpy array of shape d
        The linear model.
    true_y : numpy array of shape n
        The true output vectors. Consists of 0s and 1s.

    Returns
    -------
    loss : float
        The value of the loss.
    """
    # TODO: Fill in (Q2b)
    loss = -np.sum(true_y * np.log(sigmoid(X.T @ beta)) + (1-true_y) * np.log(sigmoid(-X.T @ beta)
    return loss
```

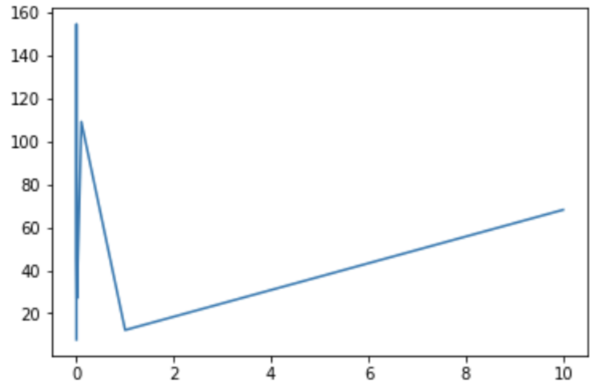
In [63]:
```
def gradient_descent(X, init_beta, true_y, loss, loss_gradient,
                     learning_rate, iterations):
    """Performs gradient descent on a given loss function and
    returns the optimized beta.

    Parameters
    ----------
    X : numpy array of shape nxd
        The feature matrix where each row corresponds to a single
        feature vector.
    init_beta : numpy array of shape d
        The initial value for the linear model.
    true_y : numpy array of shape n
        The true output vectors.
    loss : function
        The loss function we are optimizing.
    loss_gradient : function
        The gradient function that corresponds to the loss function.
    learning_rate : float
        The learning rate for gradient descent.
    iterations : int
        The number of iterations to optimize the loss for.

    Returns
    -------
    beta : numpy array of shape d
        The optimized beta.
    """
    # TODO: Fill in (Q2c)
#     beta = logistic_predict(X, init_beta)
    beta = init_beta
    for i in range(iterations):
        gradient = loss_gradient(X, beta, true_y)
        beta = beta - (gradient * learning_rate)

    return beta
```

3.

```
In [51]: # TODO: Fill in (Q2d)
def rmse(predictions, target):
    return np.sqrt(np.sum(np.power(predictions - target, 2)) / predictions.shape[0])


lambdas = [0, 10**-4, 10**-3, 10**-2, 10**-1, 1, 10]
errors = []
for l in lambdas:
    beta = regression_least_squares(boston_X_train, boston_y_train, l)
    predictions = regression_predict(boston_X_test, beta)
    error = rmse(predictions, boston_y_test)
    errors.append(error)


plt.figure()
plt.plot(lambdas, errors)
plt.show()

errors2 = []
for l in lambdas:
    beta = regression_least_squares(boston_poly_X_train, boston_y_train, l)
    predictions = regression_predict(boston_poly_X_test, beta)
    error = rmse(predictions, boston_y_test)
    errors2.append(error)

plt.figure()
plt.plot(lambdas, errors2)
plt.show()
```

4.





5. The first featurization performed better. For the polynomial features, the RMSE outputs have an off trend at smaller values of lambda, but then start to become linear.

6. The best MAE achieved is 0.6

Training model for the logistic regression dataset

In this section you will train a logistic model and evaluate it against the MAE for the Iris dataset we created above.

In [76]:
```
# TODO: Fill in (Q2f)

def indicator(p):
    return 1 if p >= 0.5 else 0

def MAE_helper(y1, y2):
    return 1 if y1 == y2 else 0

def MAE(predictions, targets):
    n = predictions.shape[0]
    return np.sum([MAE_helper(predictions[i], targets[i]) for i in range(n)]) / n

beta = gradient_descent(X=iris_X_train,
                 init_beta=np.zeros(iris_X_train.shape[1]),
                 true_y=iris_y_train,
                 loss=logistic_cross_entropy_loss,
                 loss_gradient=logistic_cross_entropy_loss_gradient,
                 learning_rate=0.5,
                 iterations=1000)

predictions = logistic_predict(iris_X_test, beta)
filtered_predictions = np.empty(predictions.shape[0])
for i in range(predictions.shape[0]):
    filtered_predictions[i] = indicator(pred)

MAE(filtered_predictions, iris_y_test)
```

Out[76]: 0.6

Problem 3

1. $E[K] = \sum_{k=0}^{\infty} kP(k) = \sum_{k=0}^{\infty} k * e^{-\lambda} \frac{\lambda^k}{k!} = \lambda * e^{-\lambda} \sum_{k=1}^{\infty} \frac{\lambda^{k-1}}{(k-1)!}$
   $E[K] = \lambda * e^{-\lambda} * e^{\lambda} = \lambda$

2. $lik(\lambda) = \prod_{i=1}^{n} P(k_i|\lambda)$

3. $\hat{\lambda_{MLE}} = \frac{d}{d\lambda} \log lik(\lambda) = \frac{d}{d\lambda} \sum_{i=1}^{n} \log(e^{-\lambda} \frac{\lambda^{k_i}}{k_i!}) = \frac{d}{d\lambda} \sum_{i=1}^{n} [k_i * \log(\lambda) - \lambda - \log k_i!]$
   $\hat{\lambda_{MLE}} = \frac{1}{\lambda} \sum_{i=1}^{n} k_i - n - 0 = 0$
   $\hat{\lambda_{MLE}} = \frac{\sum_{i=1}^{n} k_i}{n} = \bar{K}$

4. Each observation, k_i, has expectation λ and so does the sample mean, $\bar{K}$. This means the MLE is an unbiased estimator of λ.

5. $P(\lambda|k) = \frac{P(K=k|\lambda)P(\lambda)}{P(K)}$

   $P(K = k|\lambda) = \prod_{i=1}^{n} e^{-\lambda} \frac{\lambda^{k_i}}{k_i!} = \frac{\lambda^{n\bar{K}} e^{-n\lambda}}{\prod_{i=1}^{n} k_i!} \propto \lambda^{n\bar{K}} e^{-n\lambda}$
   $P(\lambda|K) \propto P(K = k|\lambda)P(\lambda) = \lambda^{n\bar{K}} e^{-n\lambda} \lambda^{\alpha-1} e^{-\beta\lambda} = \lambda^{n\bar{K}+\alpha-1} e^{-(\beta+n)\lambda}$
   Which is in the form of Gamma($n\bar{K} + \alpha$, $\beta + n$)

6. argmax $P(\lambda|K)$ = argmax $P(K = k|\lambda)P(\lambda)$
   => $\frac{d}{d\lambda} \lambda^{n\bar{K}+\alpha-1} e^{-(\beta+n)\lambda} = 0$
   $\lambda = \frac{\alpha+n\bar{K}-1}{\beta+n}$

7. The posterior is also a Gamma distribution with modified alpha and beta parameters. The MAP estimate shows this relationship in that it is the ratio of the new alpha over the new beta.