CS 6375 Assignment 3

Lakshmipriya Narayanan

Problem 1:

(a): Steps involved to use discrete Naive Bayes:

1. To use a discrete Naive Bayes we need to make the continuous features discrete. This can be done by categorizing the continuous features into a finite number of intervals. So, each interval represents a discrete category depending on the range.
2. For each feature x_i, we determine which interval it belongs to and assign a discrete value corresponding to the interval. This will help converting the continuous features to discrete.
3. Once they are transformed, we can calculate prior $P(y)$ and conditional probabilities $P(x_i|y)$.
   We use, $P(y) = \frac{\text{Number of occurrences}}{\text{Total}}$ and $P(x_i|y) = \frac{\text{Number of items in each interval for each class}}{\text{Total items in the class}}$
4. Next, we apply the Naive Bayes formula to calculate the posterior probability and then choose the class with the highest of this probability as the predicted class. Therefore, $P(y|x_1, x_2, ..., x_N) \propto P(y) \cdot \prod_{i=1}^{N} P(x_i|y)$

(b): Derivation:

As stated in the question, we assume that the features x_i are continuous and follow some distribution.
Parameters are:

1. Mean of feature x_j in class $y = \mu_{y,j}$
2. Variance of feature x_j in class $y = \sigma^2_{y,j}$
3. Prior probability $= P(y)$

Therefore, MLE for parameters $\mu_{y,j}$ and $\sigma^2_{y,j}$ are as follows:

1. The mean is estimated by $\mu_{y,j} = \frac{1}{N}\sum_{i=1}^{M} \mathbb{I}[y(i) = y] \cdot x_j(i)$ where N is the total number of data points; Indicator function $\mathbb{I}[y(i) = y]$ only chooses those data points for which class label $y(i)$ matches y (This way, only the relevant features $x_j(i)$ are included in the mean); M is the number of data points in class y.
2. The variance $\sigma^2_{y,j} = \frac{1}{N}\sum_{i=1}^{M} \mathbb{I}[y(i) = y] \cdot [x_j(i) - \mu_{y,j}]^2$
   where, the indicator function $\mathbb{I}(y(i) = y) = \begin{cases} 1, & \text{if } y(i) = y, \\ 0, & \text{otherwise.} \end{cases}$ The variance is the average squared deviation of x_j from its mean $\mu_{y,j}$
3. Prior probability $P(y) = \frac{M}{N}$ where $N =$ total number of data points and $M =$ total number of data points in class y.

To perform inference, given a new data point $X = (x_1, x_2, ...)$ we predict its class label by using the steps above. First, we calculate posterior probability for each class y:
$P(y|x) \propto P(y)\prod_{j=1}^{n} P(x_j|y)$ where $P(x_j|y)$ is the assumed probability continuous distribution. Then, we compute log-likelihood to choose the class with the highest posterior probability to assign class label that maximizes $P(y|x)$

Problem 2: SEE CODE

Problem 3: SEE CODE

Problem 4:

(a): Logistic Regression vs Decision Tree (depth 5)

Logistic Regression will have higher bias compared to a decision tree of depth= 5 because a decision tree can model more complex relationships and as model complexity increases, bias also increases. Due to this reason the decision tree will have higher variance as model complexity increases. Therefore, high bias and low variance of a logistic regression model is because of the linear relationship between predictors and logit of the response.

(b): Decision Tree vs Random Forest vs Gradient Boosted Tree

A decision tree (depth = 1) is simple but can have high variance and overfit the data and because of the trade-off, it'll have low bias.
Random forest reduces variance because they are obtained by averaging multiple trees. Therefore, this will have high bias unlike the decision tree mentioned above.
For a gradient boosted tree, bias is reduced because of sequential learning and hence will have high variance.

(c): Logistic Regression vs 1NN classifier

1NN classifier has very low bias and high variance because it assigns the class of the nearest training example to any new data point and it overfits the training data respectively.
From (a), we know that logistic regression will have high bias (linearity) and low variance; opposite of 1NN.

(d): 5NN Classifier vs 1NN Classifier

We've already seen that 1NN classifier will have low bias and high variance,
5NN classifier will have slightly higher bias compared to 1NN classifier because the decision boundary will be more smoother. This implies that it'll have lower variance compared to the 1NN classifier because it is less sensitive to noise in the training data.

(e): Depth 5 Decision Tree vs Depth 10 Decision Tree

From (a), we know that a decision tree of depth = 5 will have low bias and higher variance. But here, when it is compared with a decision tree with depth=10, the tree that is 5 units deep will have slightly higher bias and lower variance since it is not complex. The decision tree of depth = 10 will have low bias and higher variance because it fits the data better.

(f): Random Forest with 100 Trees vs Random Forest with 10 Trees

Random forest with 10 trees will have higher bias compared to random forest with 100 tress because it may not capture all complexities in the training data and a random forest with 100 trees will have lower bias because it can capture underlying patterns in the data.
The random forest with 10 trees will have higher variance because it does not fully average out the noise whereas, the random forest with 100 trees will have lower variance because it is better at averaging.

(g): AdaBoost with 10 Decision Stumps vs AdaBoost with 100 Decision Stumps.

AdaBoost with 10 decision stumps has higher bias and lower variance because it does not capture all complexities in the data and has less overfitting respectively.
AdaBoost with 100 decision stumps will have lower bias and higher variance since it is more prone to overfitting the training data.

(h): Linear Regression vs Quadratic (Polynomial degree 2) Regression.

As seen above, logistic regression has high bias and low variance. Whereas, quadratic regression has lower bias and higher variance because it can model complex relationships and is more sensitive to the training data respectively.

Problem 5:

Gradient boosting with Logistic loss:

Logistic loss function $L[f(x), y] = log[1 + e^{-y \cdot f(x)}]$ for $y \in \{-1, 1\}$

1. To minimize the loss function over the training data, set the initial prediction to a constant. For binary classification, $f_o(x) = \frac{1}{2} log(\frac{\theta}{1-\theta})$, where θ is the proportion of positive labels($y = 1$). This initialization is obtained from the log-odds transformation for logistic regression.
2. Compute residuals for each training data point by calculating the gradient of the loss function with respect to current predictions($f(x_i)$). Residuals are obtained by taking the derivative of the loss function above with respect to current predictions: $y_i^{'} = -\frac{\delta L(f(x_i), y_i))}{\delta f(x_i)} = \frac{y_i}{1+e^{y_i \cdot f(x_i)}}$. When $y_i \cdot f(x_i)$ is large and when the prediction is correct then the residual is small.
3. Then fit the regression tree $h_m(x)$ to the residuals computed above, $y_i^{'}$. This tree captures what is needed to improve the predictions of the model.
4. Update the model $f_m(x) = f_{m-1}(x) + \alpha \cdot h_m(x)$, where α is the hyperparameter learning rate and $f_{m-1}(x)$ is the current model.
5. Then, the final model is $F(x) = \text{sign}[f_M(x)]$, where $m = 1, ..., M$ is the number of iterations and the function $\text{sign}[f_M(x)]$ converts the continuous output into binary predictions $\{-1, 1\}$

Gradient boosting with Hinge loss:

Hinge loss function $L[f(x), y] = max(0, 1 - y \cdot f(x))$ for $y \in \{-1, 1\}$. We follow the exact same algorithm as the logistic loss function above but the only change is the residual computation

1. For the same number of iterations above $m = 1, ..., M$, compute residuals $y_i^{'} = \begin{cases} -y_i, & \text{if } 1 - y_i \cdot f(x_i) > 0, \\ 0, & \text{otherwise.} \end{cases}$
2. The residual is the gradient of the hinge loss function above with respect to the model's predictions.
   If $1 - y_i \cdot f(x_i) > 0$, $f(x_i)$ is within the margin, so then the residual $y_i^{'} = -y_i$ and we need to make changes in our prediction in the direction of y_i.
   If $1 - y_i \cdot f(x_i) \leq 0$, the prediction is correct (beyond the margin) and hence residual $= 0$.

Problem 6:

- ReLU activation function, $f(x) = \max(0, x)$. Then, $f^{'}(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & x \leq 0. \end{cases}$
- Logistic loss for binary classification, $L(y, \hat{y}) = C = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y})$, where y is the true label and $y \in 0, 1$ and $\hat{y}$ is the predicted probability.
  When $y = 1$, loss $= -\log\hat{y}$ and when $y = 0$, loss $= -\log(1 - \hat{y})$
- First, for a neural network with one hidden layer, the forward pass is:

  $z_j^l = \sum_k w_{jk}^l \cdot a_k^{l-1} + b_j^l$ and hence, ReLU activation is $a_j^l = max(0, z_j^l)$

  where L = number of layers, a_j^l = output of the j^{th} neuron on the l^{th} layer and, z_j^l = weighted input of the j^{th} neuron on the l^{th} layer. Here, we compute z^l for each layer,then apply ReLU for the hidden layers. For each neuron j in layer l, take outputs a_k from previous layer and multiply by corresponding weights w_{jk} and add the bias term b_j to this. This results in pre-activation value z_j.
  When we apply the ReLU activation, if $z_j^l > 0$ then output = input, whereas, if $z_j^l \leq 0$, output = 0.

- Next, for the output layer, we use the sigmoid activation function to obtain probabilities between 0and1 (Error computation).
  The final layer uses sigmoid activation instead of ReLU and converts final output to probability between 0 and 1, which is essential for binary classification.

$$\hat{y} = \sigma(z^L) = \frac{1}{(1 + e^{-z^L})}$$

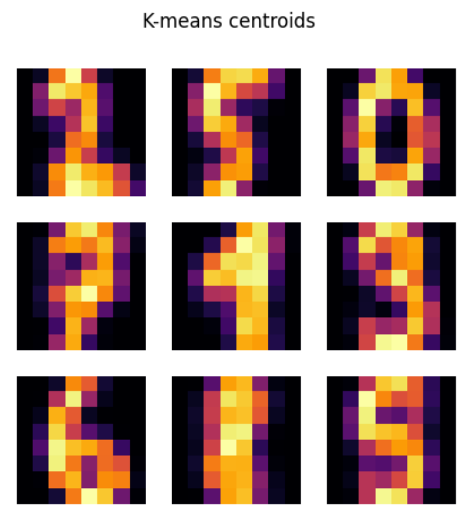
$$\begin{aligned}
\therefore \text{Outer Layer Error}, \delta^L &= \frac{\partial C}{\partial z^L} && (1)\\
&= \frac{\partial C}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^L} \text{By chain rule, take derivative of loss w.r.t output} && (2)\\
&= \left[-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \right] \cdot [\hat{y}(1-\hat{y})] \text{Multiply derivative of loss with derivative of sigmoid} && (3)\\
&= \hat{y} - y && (4)
\end{aligned}$$

- For hidden layers ($l < L$), using ReLU: $\delta^l = [w^{l+1}]^T \ \delta^{(l+1)} \cdot \text{ReLU}^{'}(z^l)$. Here, $\text{ReLU}^{'}(z^l)$ is a vector of $0s$ and $1s$ dependong on the piecewise function defined above i.e., if$z^l > 0$ is the back propagation step.
  The first term in the above expression is weighted by the connections between layers and this propagates the error from the next layer.

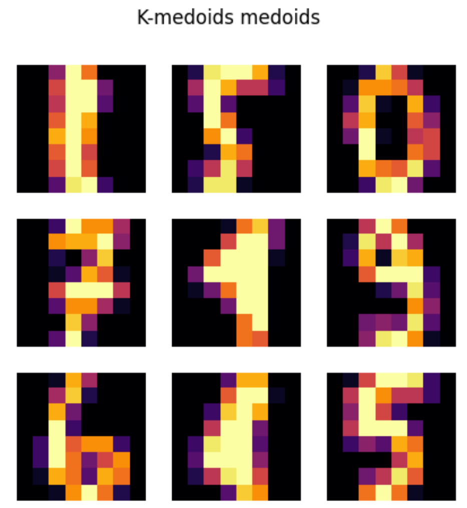
- Therefore, the gradient of weight, $\frac{\delta C}{\delta b^l_j} = \delta^l_j$ and gradient of bias, $\frac{\delta C}{\delta w^l_{jk}} = \delta^l_j \cdot a^{(l-1)}_k$ (Updating weights and bias by gradient calculation)

PROBLEM 3: Clustering

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits #Import the load_digits dataset
from sklearn.metrics import pairwise_distances_argmin

# Load the digits dataset
digits = load_digits()
dig_data = digits.data

# K-means algorithm implementation from scratch
# Input data, number of clusters and number of iterations. No Y because this is
#an unsupervised learning problem.
def kmeans(X, num_clus, max_iter = 100):
    np.random.seed(1) #set seed for reproducibility of output
    # Randomly select number of clusters with unique indices from the data
    centroids = X[np.random.choice(X.shape[0], num_clus, replace = False)]
    # Run the loop for maximum number of iterations say 100
    for _ in range(max_iter):
        # Assign labels for each data point depending on the nearest centroid by
        #computing the distance between each data point and each cetroid.
        #Return the index of the closest centrois to each data point.
        labels_y = pairwise_distances_argmin(X, centroids)

        # Update centroid: For each cluster, calculate mean of all points assigned
        #to that cluster = mean of new centroid. Convert list of new centroids to NumPy array
        new_centroids = np.array([X[labels_y == i].mean(axis=0) for i in range(num_clus)])

        # Check if the centroids have not chnaged. If yes, then convergence has occurred
        if np.all(centroids == new_centroids):
            break
        #Update centroids and return the final centroid after convergence and the cluster assignment
        centroids = new_centroids
    return centroids, labels_y

# K-medoids algorithm implementation
#Input data just like we did in k-means above
def kmedoids(X, num_clus, max_iter=100):
    np.random.seed(1)
    #Randomly select medoids and number of clusters by selecting corresponding data points
    medoids = X[np.random.choice(X.shape[0], num_clus, replace=False)]
    for _ in range(max_iter):
        # Assigns each data point to the nearest medoid just like in k-means by
        #computing the distance between each data point and each medoid. Return
        #the index of the closest medoid
        labels_y = pairwise_distances_argmin(X, medoids)

        #Update medoids
        new_medoids = [] #initialize an empty list to store the new medoids.
        #iterate over each cluster
        for i in range(num_clus):
            #select all data points assigned to the current cluster
            cluster_points = X[labels_y == i]

            #if the current cluster is empty then the current medoid remains unchanged.
            if len(cluster_points) == 0:
                new_medoids.append(medoids[i])
            # else calculate the index of the new medoid and it will be the data
            #point within the cluster that minimizes the sum of distances to
            #all other points in the cluster
            else:
                medoid_index = np.argmin(np.sum(np.linalg.norm(cluster_points[:, None] - cluster_points, axis = 2), axis = 0))
                new_medoids.append(cluster_points[medoid_index]) #add the medoid to the list above

        #Check for convergence and convert the list to NumPy array for convenience
        new_medoids = np.array(new_medoids)

        #If the medoids have remained unchnaged then the algorithm converges.
        if np.all(medoids == new_medoids):
            break
        medoids = new_medoids #update medoids to the new medoids in the curremt iteration
    #return the final medoids after convergence and the assignment for each data point.
    return medoids, labels_y

# Run k-means and k-medoids on the digits dataset with k=10 as specified in the question
k = 10
centroids_kmeans, labels_kmeans = kmeans(dig_data, k)
medoids_kmedoids, labels_kmedoids = kmedoids(dig_data, k)

# Visualize the obtained clustering by plotting

#k-means visualization
# Each plot displays one centroid. The 'figsize' parameter sets the size of the
#entire figure.
fig, axes = plt.subplots(3, 3, figsize = (5, 5))
# Iterate over each subplot (ax).
#Each centroid flattens the grid into a 1D array for easy iteration.
for ax, centroid in zip(axes.ravel(), centroids_kmeans):
    ax.imshow(centroid.reshape(8, 8), cmap = plt.cm.inferno) #display the centroid
    ax.axis('off') #remove ticks and labels
plt.suptitle('K-means centroids')
plt.show()

# Compare the performance of k-means and k-medoids clustering
def clus_loss(X, centroids, labels_y):
    return np.sum([np.linalg.norm(X[labels_y == i] - centroid) for i, centroid in enumerate(centroids)])

#Display the loss for k-means
loss_means = clus_loss(dig_data, centroids_kmeans, labels_kmeans)
print(f"K-means clustering loss: {loss_means: .4f}")

#k-medoids visualization
# Same strategy as above in k-means visualization
fig, axes = plt.subplots(3, 3, figsize = (5, 5))
for ax, medoid in zip(axes.ravel(), medoids_kmedoids):
    ax.imshow(medoid.reshape(8, 8), cmap = plt.cm.inferno)
    ax.axis('off')
plt.suptitle('K-medoids medoids')
plt.show()

#Display the loss for k-medoids
loss_medoids = clus_loss(dig_data, medoids_kmedoids, labels_kmedoids)
print(f"K-medoids clustering loss: {loss_medoids: .4f}")
```



```
K-means clustering loss:  3357.6981
```



```
K-medoids clustering loss:  3913.0012
```

```
Lower loss values indicate that data points are closer to the center of their
clusters implting better cluster performance. Here, the K-means algorithm has
lower loss than K-Medoids algorithm, implying that for this dataset, the
k-means algorithm has achieved better clustering.
```

Lower loss values indicate that data points are closer to the center of their clusters implting better cluster performance. Here, the K-means algorithm has lower loss than K-Medoids algorithm, implying that for this dataset, the k-means algorithm has achieved better clustering.

PROBLEM 4: Logistic Regression

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.metrics import accuracy_score

# Load Breast Cancer dataset from sklearn.datasets
data = load_breast_cancer()
X = data.data
y = data.target

# Splitting data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

# Standardize features using
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

class LogitReg:
    # Initialize logistic regression model with parameters learning rate for
    # gradient descent, iterations and regularization as specified in the
    # question.
    def __init__(self, learn_rate = 0.01, num_iter = 1000, regn = 0):
        self.learn_rate = learn_rate
        self.num_iter = num_iter
        self.regn = regn

    # Compute sigmoid function to map any real-valued number into (0, 1)
    # interval so that it satisfies the assumptions of the logistic regression
    # model.
    def sigmoid_func(self, z):
        return 1 / (1 + np.exp(-z))

    # Fit the logistic regression model on the features and the labels
    def fit(self, X, y):
        self.m, self.n = X.shape    # m = number of samples; n = number of features
        self.weight = np.zeros(self.n)   # Initialize the weights to zero
        self.bias = 0  # Initialize bias to 0

        # Perform gradient descent to optimize the weights and bias
        for i in range(self.num_iter):
            lin_model = np.dot(X, self.weight) + self.bias   # Compute linear combination of inputs, weights and bias.
            y_pred = self.sigmoid_func(lin_model)   # Apply sigmoid function to obtain predicted probabilities.

            # Compute gradients for weights and bias with regularization term
            dw = (1 / self.m) * np.dot(X.T, (y_pred - y)) + (self.regn / self.m) * self.weight
            db = (1 / self.m) * np.sum(y_pred - y)

            # Update weights and bias
            self.weight -= self.learn_rate * dw
            self.bias -= self.learn_rate * db

    # Make predictions on new features
    def predict(self, X):
        # Compute linear combination and apply sigmoid function and return
        # corresponding labels.
        lin_model = np.dot(X, self.weight) + self.bias
        y_pred = self.sigmoid_func(lin_model)
        return [1 if i > 0.5 else 0 for i in y_pred]

# Train logistic regression with regularization = 0
model = LogitReg(learn_rate = 0.01, num_iter = 1000, regn = 0)
model.fit(X_train_scaled, y_train)

# Mkae predictions on training and test sets
y_train_pred = model.predict(X_train_scaled)
y_test_pred = model.predict(X_test_scaled)

# Calculate accuracy on training and test sets
train_acc = accuracy_score(y_train, y_train_pred)
print(f"Train Accuracy = {train_acc: .4f}")
test_acc = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy = {test_acc: .4f}")

# Check if we are underfitting or overfitting using threshold value of 0.8
if train_acc < 0.8 and test_acc < 0.8:
    print("The model is underfitting.")
elif train_acc > 0.8 and test_acc < 0.8:
    print("The model is overfitting.")
else:
    print("The model is neither underfitting nor overfitting.")
```

```
Train Accuracy =  0.9824
Test Accuracy =  0.9825
The model is neither underfitting nor overfitting.
```

Notice that both training and test accuracies are very high, indiacting that this model is performing well on this breast cancer dataset indicating that the model is effective in predicting whether a tumor is malignant or benign and the model learned the important features that distiguish between the two classes.

```
# Since model is not overfitting, create polynomial features and train logistic
# regression with zero regularization.

# If not overfitting then create polynomial features
if not (train_acc > 0.8 and test_acc < 0.8):
    poly_degs = [2, 3, 4]  #Use polynomials of degree 2-4
    # For each degree, create a polynomial feature and transform training data
    # and test data to include polynomial features using the PolynomialFeatures
    # package.
    for deg in poly_degs:
        poly = PolynomialFeatures(deg)
        X_poly_train = poly.fit_transform(X_train_scaled)
        X_poly_test = poly.transform(X_test_scaled)

        # Train the new logistic regression model with below specified
        # parameters and fit using polynomial features.
        model_poly = LogitReg(learn_rate = 0.01, num_iter = 1000, regn = 0)
        model_poly.fit(X_poly_train, y_train)

        # Make predictions for training and test labels
        y_poly_train_pred = model_poly.predict(X_poly_train)
        y_poly_test_pred = model_poly.predict(X_poly_test)

        # Calculate accuracy for above made predictions
        train_poly_accuracy = accuracy_score(y_train, y_poly_train_pred)
        test_poly_accuracy = accuracy_score(y_test, y_poly_test_pred)

        # Display results
        print(f"Degree of polynomial = {deg}")
        print(f"Train Accuracy with Polynomial Features = {train_poly_accuracy: .4f}")
        print(f"Test Accuracy with Polynomial Features = {test_poly_accuracy: .4f}")
        print(f"-------------------------------------------------------------------------------------------")

        # Check for overfitting
        if train_poly_accuracy > 0.8 and test_poly_accuracy < 0.8:
            print("The model is overfitting with polynomial features of degree", deg)
            break

    # Add regularization to reduce overfitting
    # Define a lost of regularizations to test for decrease in overfitting
    regns = [0.01, 0.1, 1]

    # For each regularization value, initialize the logistic regression model
    # with current regularization with respective parameters and fit the model
    # until we overfit (Here overfitting happens at degree 4 so we stop here and
    # start adding regularization)
    for reg in regns:
        model_reg = LogitReg(learn_rate=0.01, num_iter=1000, regn=reg)
        model_reg.fit(X_poly_train, y_train)

        # Make predictions for the labels in training and test data using the
        # above trained model and calculate accuracies respectively.
        y_reg_train_pred = model_reg.predict(X_poly_train)
        y_reg_test_pred = model_reg.predict(X_poly_test)
        train_reg_acc = accuracy_score(y_train, y_reg_train_pred)
        test_reg_acc = accuracy_score(y_test, y_reg_test_pred)

        # Display results
        print(f"Regularization = {reg}")
        print(f"Train Accuracy with regularization = {train_reg_acc: .4f}")
        print(f"Test Accuracy with regularization = {test_reg_acc: .4f}")
        print(f"-------------------------------------------------------------------------------------------")
```

```
Degree of polynomial = 2
Train Accuracy with Polynomial Features =  0.9824
Test Accuracy with Polynomial Features =  0.9561
-------------------------------------------------------------------------------------------
<ipython-input-7-5998f18eda91>:33: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
Degree of polynomial = 3
Train Accuracy with Polynomial Features =  0.9978
Test Accuracy with Polynomial Features =  0.9649
-------------------------------------------------------------------------------------------
<ipython-input-7-5998f18eda91>:33: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
Degree of polynomial = 4
Train Accuracy with Polynomial Features =  1.0000
Test Accuracy with Polynomial Features =  0.9561
-------------------------------------------------------------------------------------------
<ipython-input-7-5998f18eda91>:33: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
Regularization = 0.01
```

```
Train Accuracy with regularization =  1.0000
Test Accuracy with regularization =  0.9561
------------------------------------------------------------------------------------------
<ipython-input-7-5998f18eda91>:33: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
Regularization = 0.1
Train Accuracy with regularization =  1.0000
Test Accuracy with regularization =  0.9561
------------------------------------------------------------------------------------------
<ipython-input-7-5998f18eda91>:33: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
Regularization = 1
Train Accuracy with regularization =  1.0000
Test Accuracy with regularization =  0.9561
------------------------------------------------------------------------------------------
```

Once we added polynomial features, our training and testing accuracies were again high but for a fourth degree polynomial we attained a training accuracy of 1 implying an overfit. Also, as we increased the degrees of the polynomial we noticed that despite the model performing well, since test accuarcy keeps decreasing and training accuarcy keeps increasing, we increasingly overfit our model. So we stop here and start adding regularization. Once, we added regularization we still severely overfit on out training data for all values of regularization. Therefore, regularization did not reduce overfitting; in fact, it stayed the same when we increased regularization.

In conclusion, this logistic regression model shows signs of overfitting, especially with higher degree polynomial features despite regularization. The consistent perfect training accuracy may indicate that the model is too complex.