Homework 3

Bruno Morgado
RIN: 661995422

Question 1: Image clustering through K-Means

In [ ]:
```
from sklearn.cluster import KMeans
from skimage import io
import numpy as np
from sklearn.metrics import silhouette_score
```

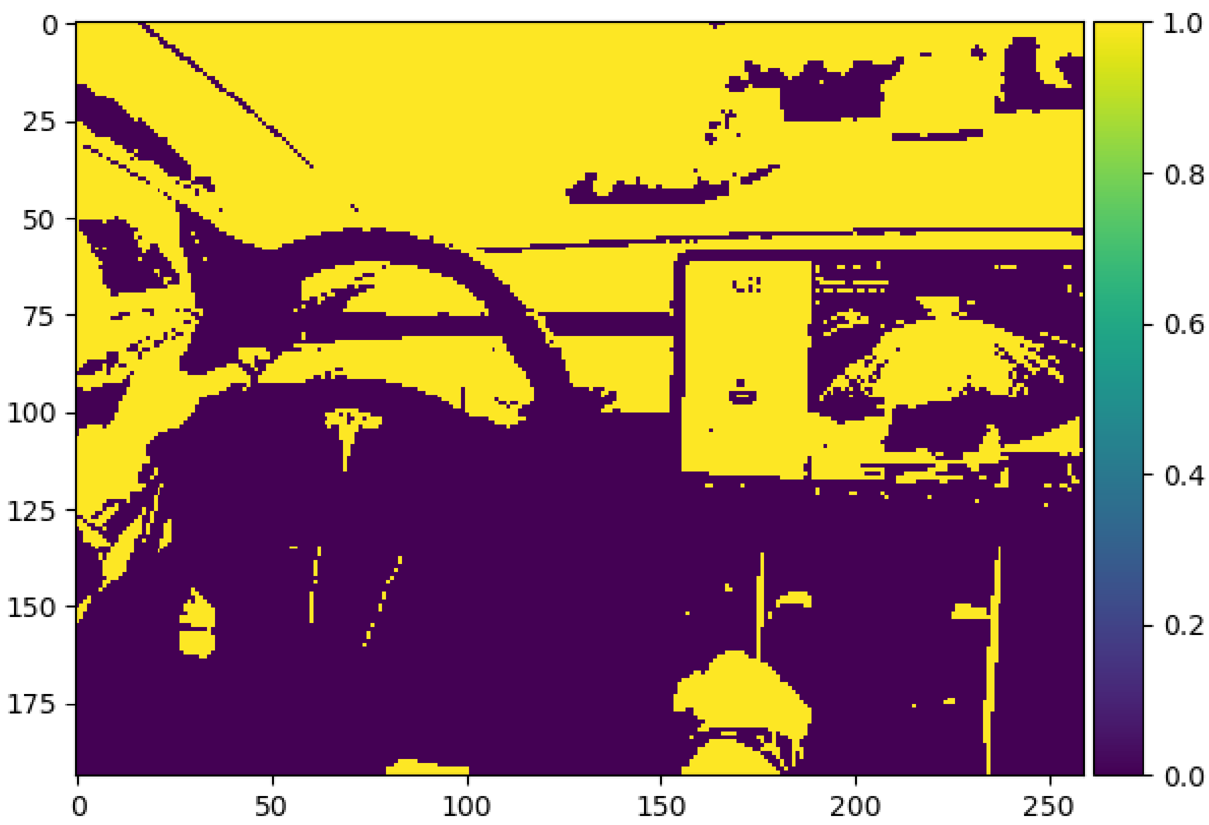
In [ ]:
```
# load image
image = io.imread('seg2.jpg')
io.imshow('seg2.jpg')
```

Out[ ]: <matplotlib.image.AxesImage at 0x1c4e4fa4e80>

In [ ]:
```
# reshape image to a 2D array of pixels
pixels = image.reshape(-1,3)

# initialize KMeans model
kmeans = KMeans(n_clusters=2, random_state= 120)

# fit the model to the pixels
```

```
kmeans.fit(pixels)

# get the labels for each pixel
labels = kmeans.labels_

# reshape the labels back to the original image shape
segmented_image = labels.reshape(image.shape[0], image.shape[1])

# display the segmented image
io.imshow(segmented_image)
io.show()

score = silhouette_score(pixels, kmeans.fit_predict(pixels))

print("silhouette score: ", score)
```

```
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\sklearn\clust
er\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10
to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\skimage\io\_p
lugins\matplotlib_plugin.py:150: UserWarning: Low image data range; displaying image
with stretched contrast.
  lo, hi, cmap = _get_display_range(image)
```



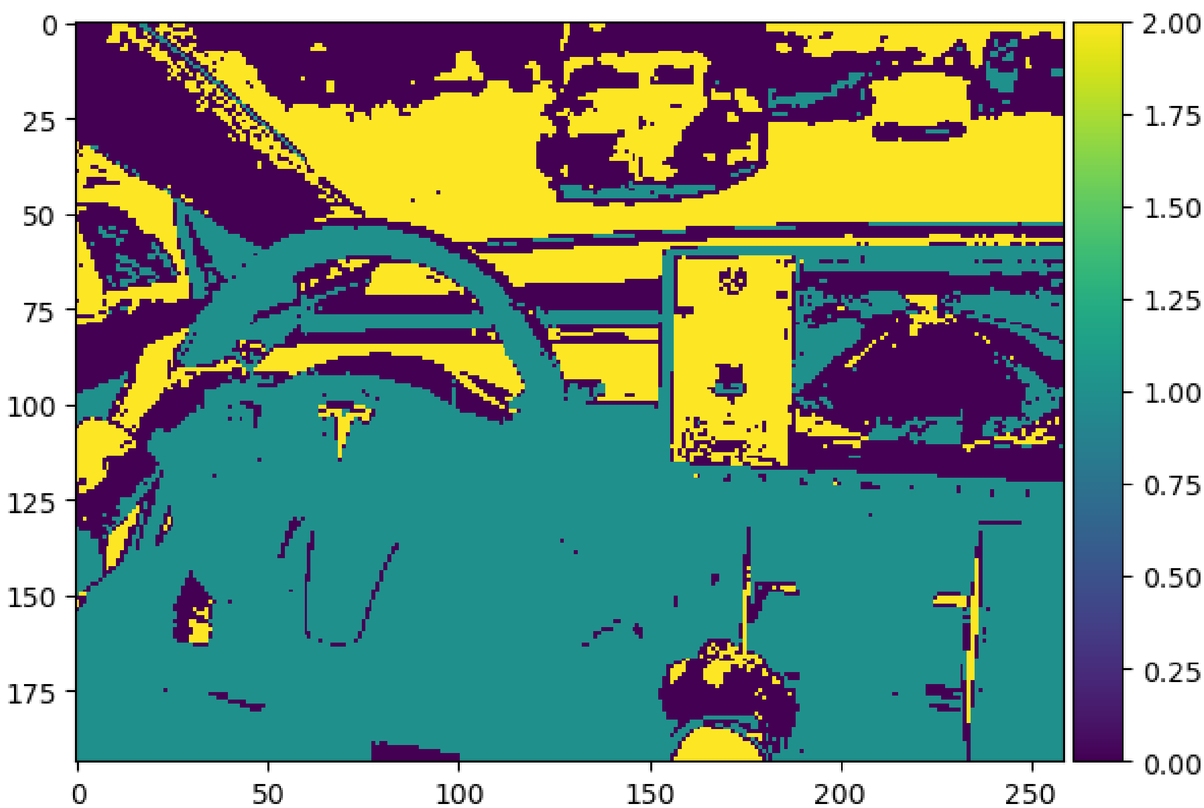
```
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\sklearn\clust
er\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10
to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
silhouette score:  0.6593767748905157
```

In [ ]:

```
# reshape image to a 2D array of pixels
pixels = image.reshape(-1,3)
```

```python
# initialize KMeans model
kmeans = KMeans(n_clusters=3, random_state= 120)

# fit the model to the pixels
kmeans.fit(pixels)

# get the labels for each pixel
labels = kmeans.labels_

# reshape the labels back to the original image shape
segmented_image = labels.reshape(image.shape[0], image.shape[1])

# display the segmented image
io.imshow(segmented_image)
io.show()

score = silhouette_score(pixels, kmeans.fit_predict(pixels))

print("silhouette score: ", score)
```

```
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\sklearn\clust
er\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10
to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\skimage\io\_p
lugins\matplotlib_plugin.py:150: UserWarning: Low image data range; displaying image
with stretched contrast.
  lo, hi, cmap = _get_display_range(image)
```

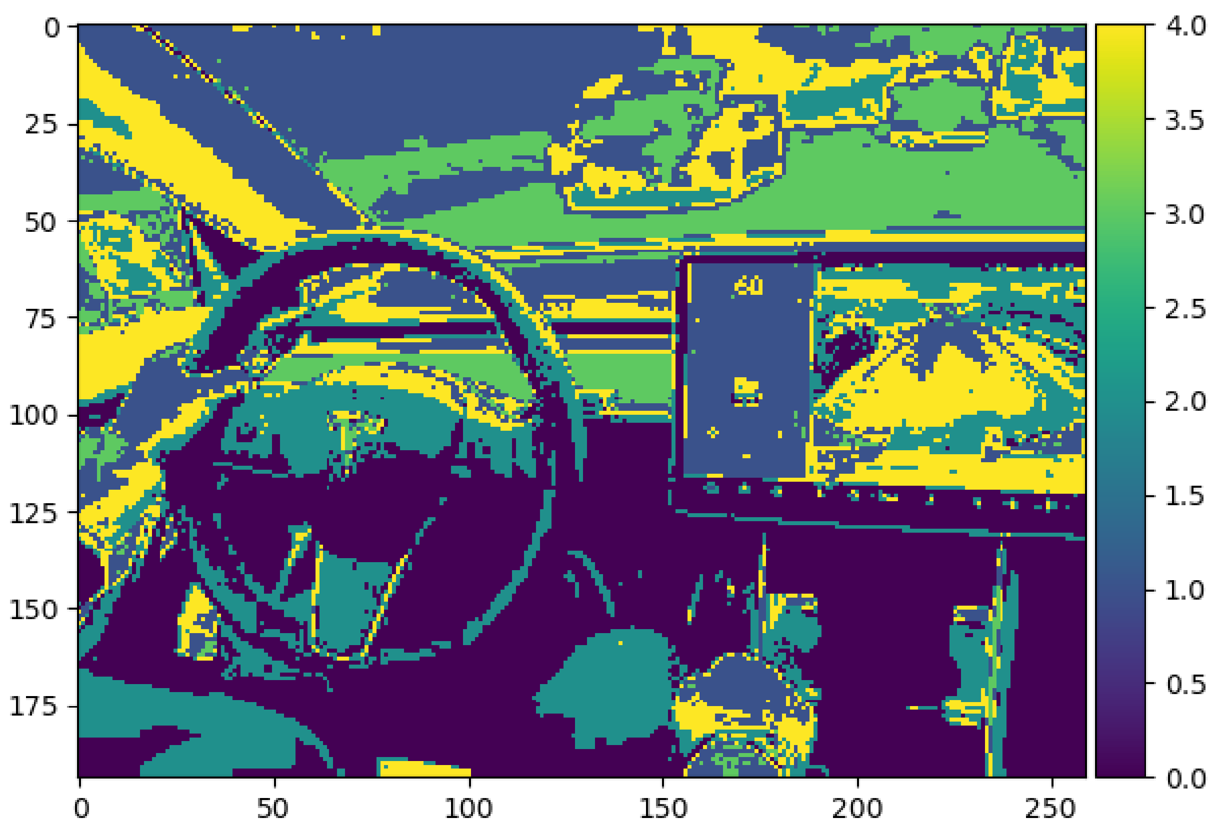


```
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\sklearn\clust
er\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10
to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
silhouette score:  0.5718655234925686
```

In the current implementation similar pixels are clustered together and are divided into 3 different classes. there is a very coarse division of the objects.

In [ ]:
```
# initialize KMeans model
kmeans = KMeans(n_clusters=5, random_state=120)

# fit the model to the pixels
kmeans.fit(pixels)

# get the labels for each pixel
labels = kmeans.labels_

# reshape the labels back to the original image shape
segmented_image = labels.reshape(image.shape[0], image.shape[1])

# display the segmented image
io.imshow(segmented_image)
io.show()

score = silhouette_score(pixels, kmeans.fit_predict(pixels))

print("silhouette score: ", score)
```

```
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\sklearn\clust
er\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10
to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\skimage\io\_p
lugins\matplotlib_plugin.py:150: UserWarning: Low image data range; displaying image
with stretched contrast.
  lo, hi, cmap = _get_display_range(image)
```



```
c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\site-packages\sklearn\clust
er\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10
to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
silhouette score:  0.5287130514225519
```

in the case where the clusters were divided in to 5 as seen above, more features can be identified, but there is a loss of generalizeability. The wheel can be identified, but it has 3 different clusters on it, for example.

Question 2: Linear Model using mini-batch gradient descent.

In this model we want to optimize a linear model by the use of mini-batch gradient descent. We first must establish the cost function, the number of regressors, the learning rate, the number of points used to calculate the function and the size of the batches. The class below was written for gradient descent optimization of linear models. The implementation for the gradient allows for both the use of an analytical loss gradient or a numerical loss gradient (this allows for the loss function to not be the RSS based function).

In [ ]:
```python
import numpy as np
from matplotlib import pyplot as plt
debug = False

class BM_linear():
    """ My implementation of a linear model suite with batch, mini-batch and stochasti
    """
    def __init__(self, X, y):
        """_summary_

        Args:
            X (_type_): _description_
            y (_type_): _description_

        Operations:
            K - is the number of regressor used in the
        """
        self.X = X
        self.y = y
        self.K = np.shape(X)[1]
        self.beta = np.random.rand(self.K,)
        self.J = []


    def cost_GD(self, beta):
        """_summary_

        Args:
            beta (_type_): _description_

        Returns:
            _type_: _description_
        """
        u = self.y - self.X @ beta
```

```
        RSS = u.dot(u)
        return 1/(2*np.size(self.y)) * RSS

    def cost_mBGD(self, indeces, beta):
        """_summary_

        Args:
            indeces (_type_): _description_
            beta (_type_): _description_

        Returns:
            _type_: _description_
        """
        y_hat = (self.X[indeces,:] @ beta)
        u = self.y[indeces] - y_hat
        RSS = u.dot(u)
        if debug:
            print("u: ", u)
            print("RSS: ", RSS)
            print("y: ", self.y[indeces])
            print("y_hat: ", self.X[indeces,:] @ beta)

        return 1/(2*np.size(indeces)) * RSS

    def analitic_grad_mBGD(self, indeces, beta):
        """_summary_

        Args:
            indeces (_type_): _description_
            beta (_type_): _description_

        Returns:
            _type_: _description_
        """
        y_hat = (self.X[indeces,:] @ beta)
        u = self.y[indeces] - y_hat
        return -1/np.size(indeces) * self.X[indeces,:].T @ u

    def analitic_grad_GD(self, beta):
        """_summary_

        Args:
            indeces (_type_): _description_
            beta (_type_): _description_

        Returns:
            _type_: _description_
        """
        y_hat = (self.X @ beta)
        u = self.y - y_hat
        return -1/np.size(self.y ) * self.X.T @ u


    def eval_cost_mBGD(self, indeces):
        """_summary_

        Args:
            batch_size (_type_): _description_

        Returns:
```

```
            _type_: _description_
        """

        J = self.cost_mBGD(indeces, self.beta)
        grad_J = self.analitic_grad_mBGD(indeces, self.beta)
        if debug:
            print("J: ", J)
            print("grad(J): ", grad_J)
            print("beta: ",self.beta, " of shape: ", np.shape(self.beta))
        return J, grad_J

    def eval_cost_mBGD_complex(self, indeces):
        """_summary_

        Args:
            batch_size (_type_): _description_

        Returns:
            _type_: _description_
        """

        J = self.cost_mBGD(indeces, self.beta)
        if debug:
            print("J: ", J)


        # To calculate the gradient in J using the complex step method
        grad_J = []
        for i in range(self.K):
            print(i)
            h = 1e-16
            beta = self.beta.astype('complex')

            beta[i] += h*1j
            dJ = np.imag(self.cost_mBGD(indeces, beta))/h
            if debug:
                print("dJ: ", dJ)
            grad_J.append(dJ)

        grad_J = np.array(grad_J)
        if debug:
            print("grad(J): ", grad_J)
            print("beta: ",beta, " of shape: ", np.shape(beta))

        return J, grad_J

    def eval_cost_GD(self):
        """_summary_

        Args:
            batch_size (_type_): _description_

        Returns:
            _type_: _description_
        """

        J = self.cost_GD(self.beta)
        grad_J = self.analitic_grad_GD(self.beta)

        if debug:
```

```
            print("J: ", J)
            print("grad(J): ", grad_J)
            print("beta: ",self.beta, " of shape: ", np.shape(self.beta))
        return J, grad_J

    def fit_mBGD(self, batch_size, tol, max_iter, learn_rate):

        i = 0
        while i < max_iter or np.linalg.norm(grad_J*learn_rate)<tol:
            indeces = self.circular_indeces(np.size(self.y), batch_size, i) # this lir
            if debug:
                print("indeces: ", indeces)
            i+=1
            J, grad_J = self.eval_cost_mBGD(indeces)
            self.beta -= learn_rate*grad_J
            self.J.append(J)

        return self.beta, self.J

    def fit_SmBGD(self, batch_size, tol, max_iter, learn_rate):

        i = 0
        while i < max_iter or np.linalg.norm(grad_J*learn_rate)<tol:
            indeces = np.random.choice(np.size(self.y), batch_size) # this line choses
            if debug:
                print("indeces: ", indeces)
            i+=1
            J, grad_J = self.eval_cost_mBGD(indeces)
            self.beta -= learn_rate*grad_J
            self.J.append(J)

        return self.beta, self.J

    def fit_GD(self, tol, max_iter, learn_rate):

        i = 0
        while i < max_iter or np.linalg.norm(grad_J*learn_rate)<tol:
            i+=1
            J, grad_J = self.eval_cost_GD()
            self.beta -= learn_rate*grad_J
            self.J.append(J)

        return self.beta, self.J

    ## Helper Functions
    def circular_indeces(self, vec_size, batch_size, i):

        number_of_batches = int(vec_size/batch_size)
        batch_index = i%number_of_batches

        if batch_index < number_of_batches-1:
            lower_bound = batch_index*batch_size
            upper_bound = (batch_index+1)*batch_size
            return np.arange(lower_bound,upper_bound)

        elif vec_size%batch_size == 0 and batch_index == number_of_batches-1:
            lower_bound = batch_index*batch_size
            upper_bound = (batch_index+1)*batch_size
            return np.arange(lower_bound,upper_bound)
```

```
        elif vec_size%batch_size > 0 and batch_index == number_of_batches-1:
            lower_bound = (batch_index+1)*batch_size
            return np.arange(lower_bound,vec_size)
```

Prepare data and initialize model

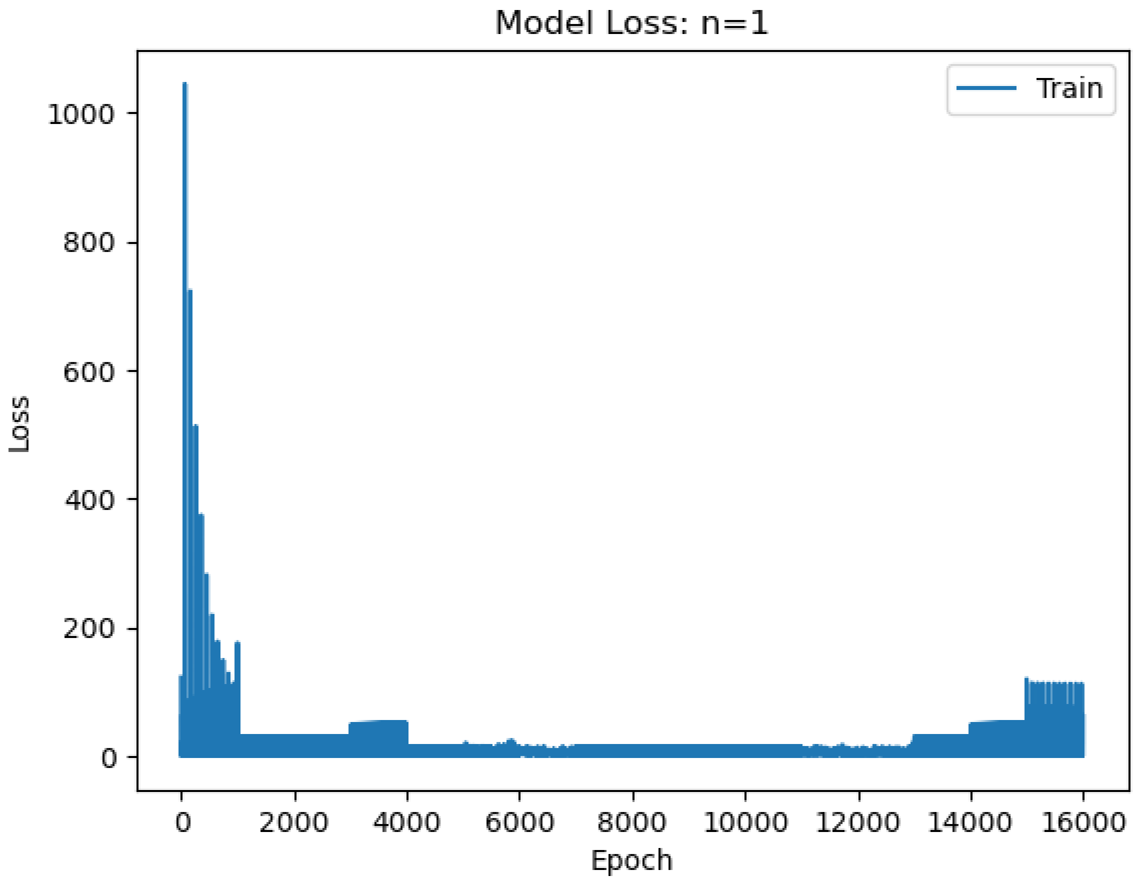
In this data set I am extracting the x and y data and adding a bias term as well.

In [ ]:
```
data = np.loadtxt('housing_prices.txt', delimiter=',', skiprows=1)

X = data[:,0]
const = np.ones([np.shape(X)[0], 1])
X_1d = np.append(const, X)
X = np.reshape(X_1d, (np.shape(X)[0], 2), 'F')

y = data[:,1]


tol = 1e-4
max_iter = 2000
learn_rate = 0.01

model = BM_linear(X,y)
```

prensent data and iterations...

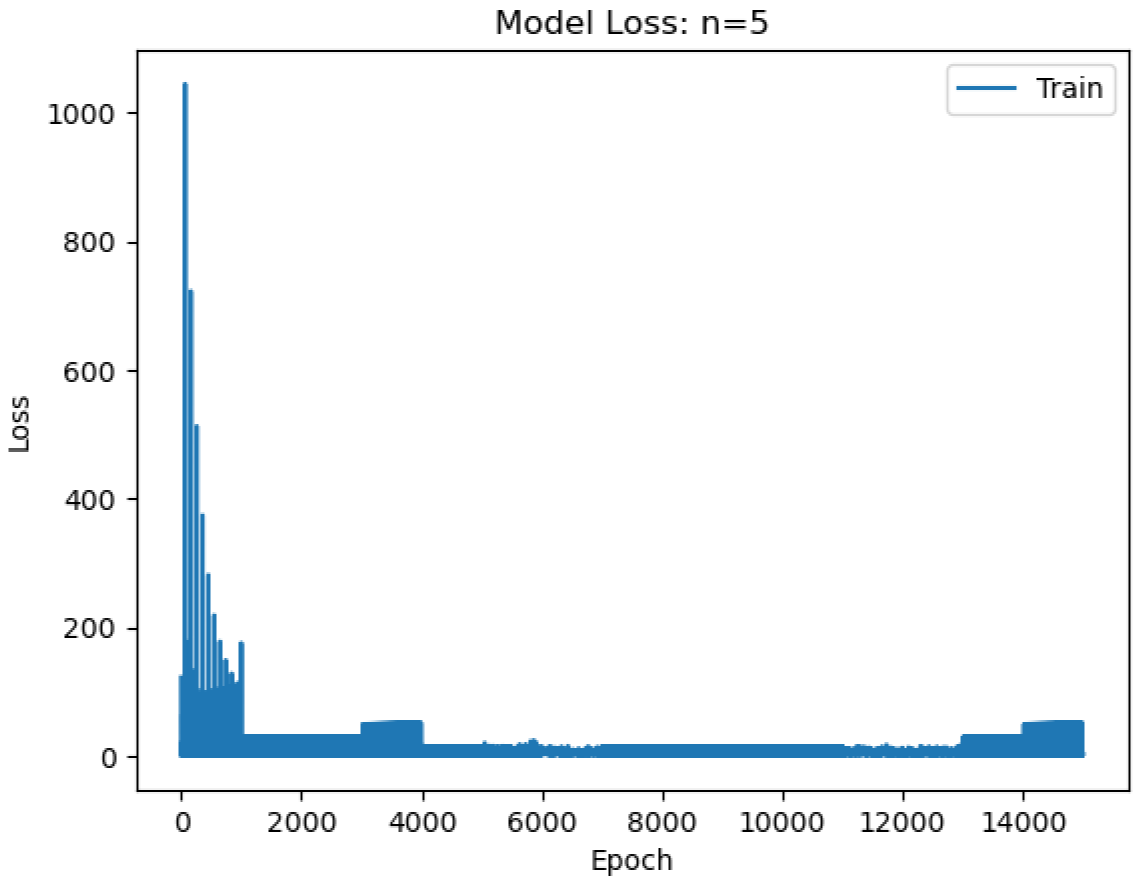
In [ ]:
```
batch_size = 1
beta, cost = model.fit_mBGD(batch_size, tol, max_iter, learn_rate)
beta_analytic = np.linalg.inv(X.T@X)@X.T@y

print("Model Parameters: ", beta)
print("Analytical Parameters: ", beta_analytic)
plt.plot(cost)
plt.title('Model Loss: n='+ str(batch_size))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend('Train', loc='upper right')
plt.show()
```

```
Model Parameters:  [-3.91239088 -0.71204413]
Analytical Parameters:  [-3.89578088  1.19303364]
```



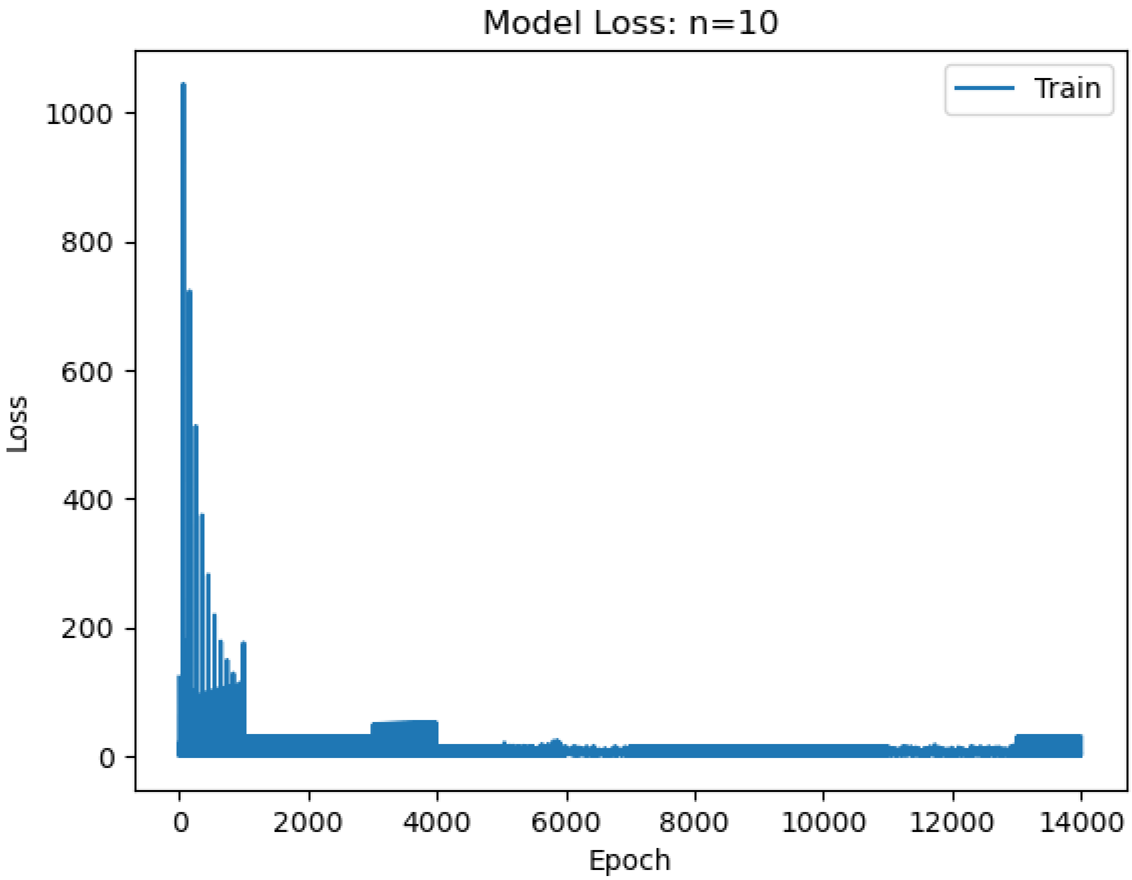
In [ ]:
```
batch_size = 5
beta, cost = model.fit_mBGD(batch_size, tol, max_iter, learn_rate)
beta_analytic = np.linalg.inv(X.T@X)@X.T@y

print("Model Parameters: ", beta)
print("Analytical Parameters: ", beta_analytic)
plt.plot(cost)
plt.title('Model Loss: n='+ str(batch_size))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend('Train', loc='upper right')
plt.show()
```

```
Model Parameters:  [-4.03555844  1.00891947]
Analytical Parameters:  [-3.89578088  1.19303364]
```



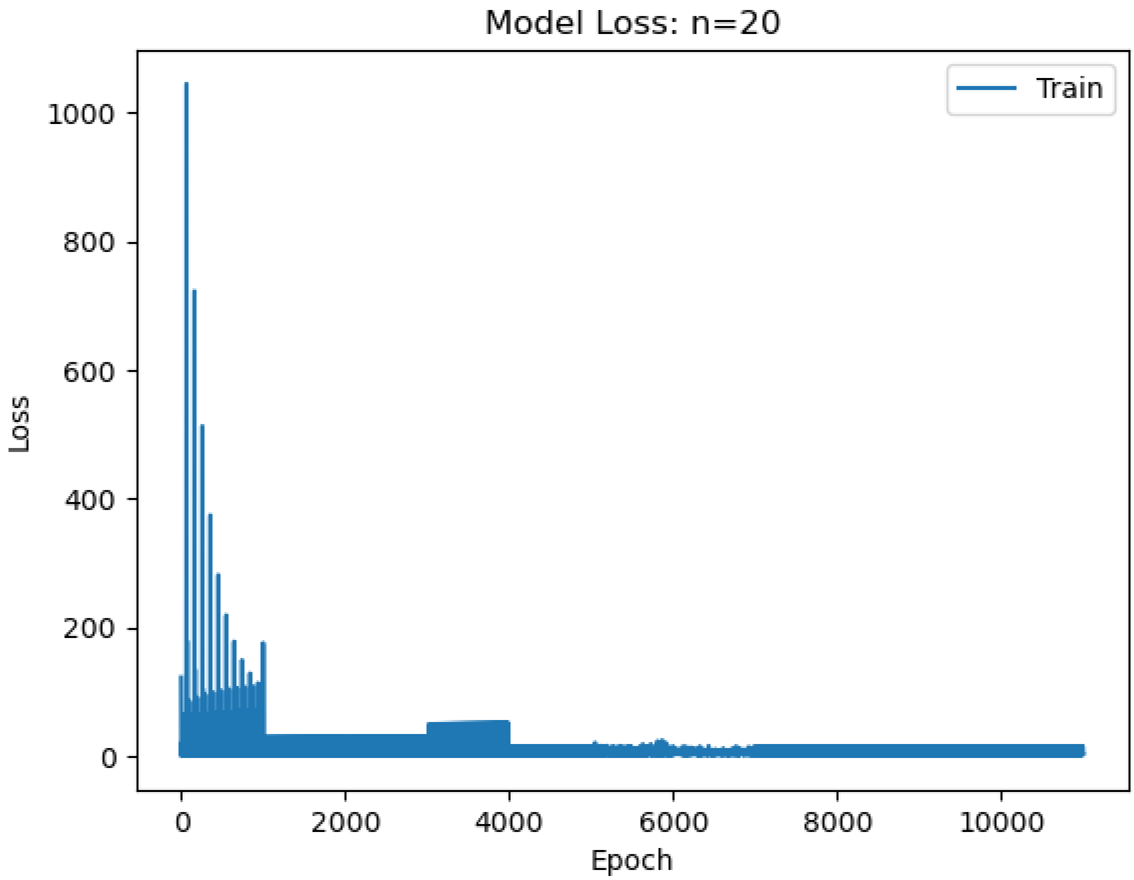
In [ ]:
```
batch_size = 10
beta, cost = model.fit_mBGD(batch_size, tol, max_iter, learn_rate)
beta_analytic = np.linalg.inv(X.T@X)@X.T@y

print("Model Parameters: ", beta)
print("Analytical Parameters: ", beta_analytic)
plt.plot(cost)
plt.title('Model Loss: n='+ str(batch_size))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend('Train', loc='upper right')
plt.show()
```

```
Model Parameters:  [-3.43068042  1.43011266]
Analytical Parameters:  [-3.89578088  1.19303364]
```



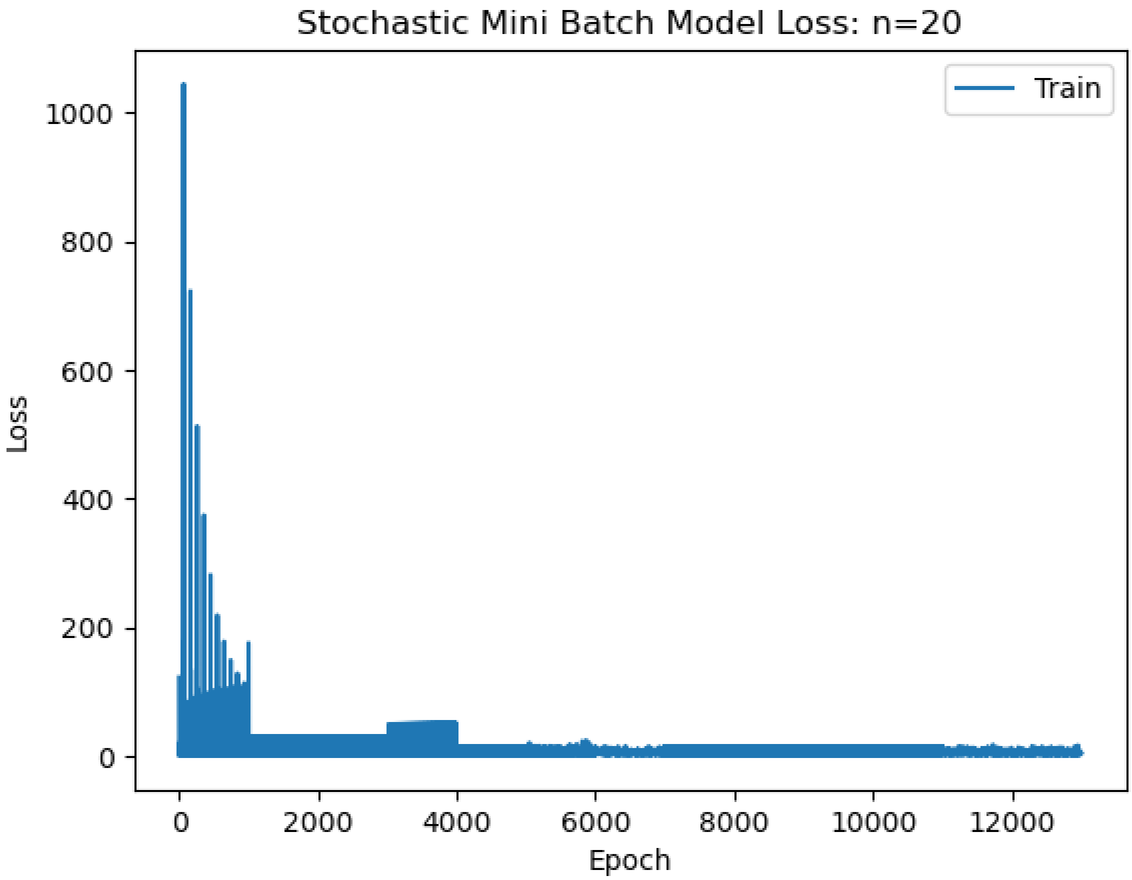
In [ ]:
```
batch_size = 20
beta, cost = model.fit_mBGD(batch_size, tol, max_iter, learn_rate)
beta_analytic = np.linalg.inv(X.T@X)@X.T@y

print("Model Parameters: ", beta)
print("Analytical Parameters: ", beta_analytic)
plt.plot(cost)
plt.title('Model Loss: n='+ str(batch_size))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend('Train', loc='upper right')
plt.show()
```

```
Model Parameters:  [-3.11738411  0.96614116]
Analytical Parameters:  [-3.89578088  1.19303364]
```



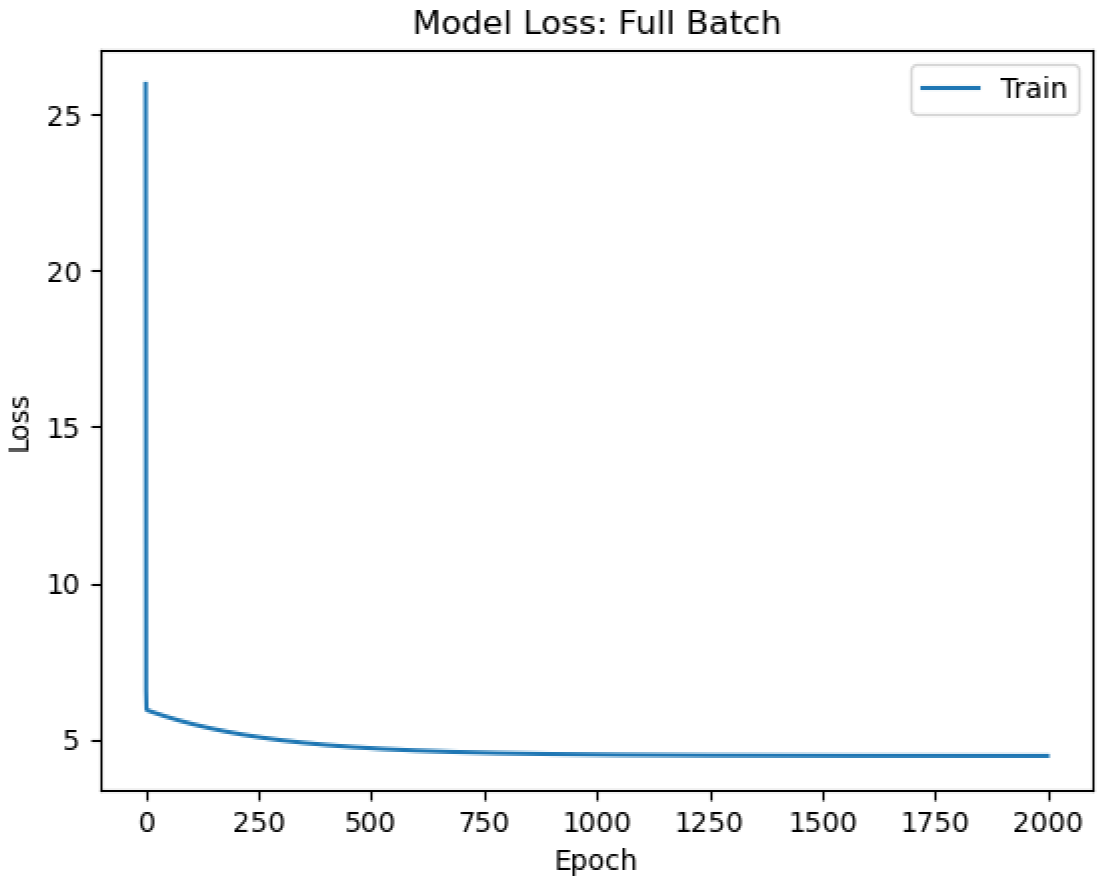
In [ ]:
```
batch_size = 20
beta, cost = model.fit_SmBGD(batch_size, tol, max_iter, learn_rate)
beta_analytic = np.linalg.inv(X.T@X)@X.T@y

print("Model Parameters: ", beta)
print("Analytical Parameters: ", beta_analytic)
plt.plot(cost)
plt.title('Stochastic Mini Batch Model Loss: n='+ str(batch_size))
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend('Train', loc='upper right')
plt.show()
```

```
Model Parameters:  [-3.85491014  1.17401418]
Analytical Parameters:  [-3.89578088  1.19303364]
```



In [ ]:
```
beta, cost = model.fit_GD(tol, max_iter, learn_rate)
beta_analytic = np.linalg.inv(X.T@X)@X.T@y

print("Model Parameters: ", beta)
print("Analytical Parameters: ", beta_analytic)
plt.plot(cost)
plt.title('Model Loss: Full Batch')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
Model Parameters:  [-3.78652055  1.18205726]
Analytical Parameters:  [-3.89578088  1.19303364]
```



Most of the descent methods reach a good estimate for the actual parameters of the model. In fact the most effective estimate for the analytic values was the stochastic mini-batch descent method in this case. That is if we are using limited information it is best to minimize the model by increasing the sample variances. Otherwise a full batch approach actually got really good results, but not exactly the final value as expected.

Question 3: Logistic Regression for cancer data

In this data set I will be using a logistic regression model from scikit-learn in order to run a classification exercise on beast cancer data. I will implement a cross-validation method to progressively reduce the number of features to find the best two features to perform the classification.

In [ ]:
```
### Import Module
from sklearn import datasets
from sklearn import model_selection
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt


### Prepare data
data = datasets.load_breast_cancer()
```

```python
print("original feature names: \n", data['feature_names'],"\n")
print("Number of features: ", len(data['feature_names']))
X = data['data']
y = data['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=
```

```
original feature names:
 ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']

Number of features:  30
```

In [ ]:

```python
### Train model
lr = LogisticRegression(solver='lbfgs', max_iter=5000)
rfe = RFE(lr, n_features_to_select=2)
rfe.fit(X_train, y_train)
print("Feature selection active?\n ", rfe.support_)
feat_names = data['feature_names']
print("\nActive features: ", feat_names[rfe.support_])
```

```
Feature selection active?
  [False False False False False False False False False False False False
 False False False False False False False False False False False False
 False  True  True False False False]

Active features:  ['worst compactness' 'worst concavity']
```

In [ ]:

```python
print(classification_report(y_test, rfe.predict(X_test)))
```

```
              precision    recall  f1-score   support

           0       0.82      0.63      0.71        59
           1       0.83      0.93      0.87       112

    accuracy                           0.82       171
   macro avg       0.82      0.78      0.79       171
weighted avg       0.82      0.82      0.82       171
```

In [ ]:

```python
print(classification_report(y_train, rfe.predict(X_train)))
```

```
              precision    recall  f1-score   support

           0       0.86      0.70      0.77       153
           1       0.83      0.93      0.88       245

    accuracy                           0.84       398
   macro avg       0.85      0.81      0.83       398
weighted avg       0.84      0.84      0.84       398
```

The model has trained for a 70-30 split and has an accuracy of 82% on the test data which was similar to the accuracy of the training data suggesting the model was not overfit. The two highest explanation features were the worst compatness and worst concavity features

Question 4: 2 Neuron neural network to predict housing prices

In [ ]:
```
import tensorflow as tf
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers
from tensorflow.python.keras.optimizers import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
WARNING:tensorflow:From c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\sit
e-packages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy
is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

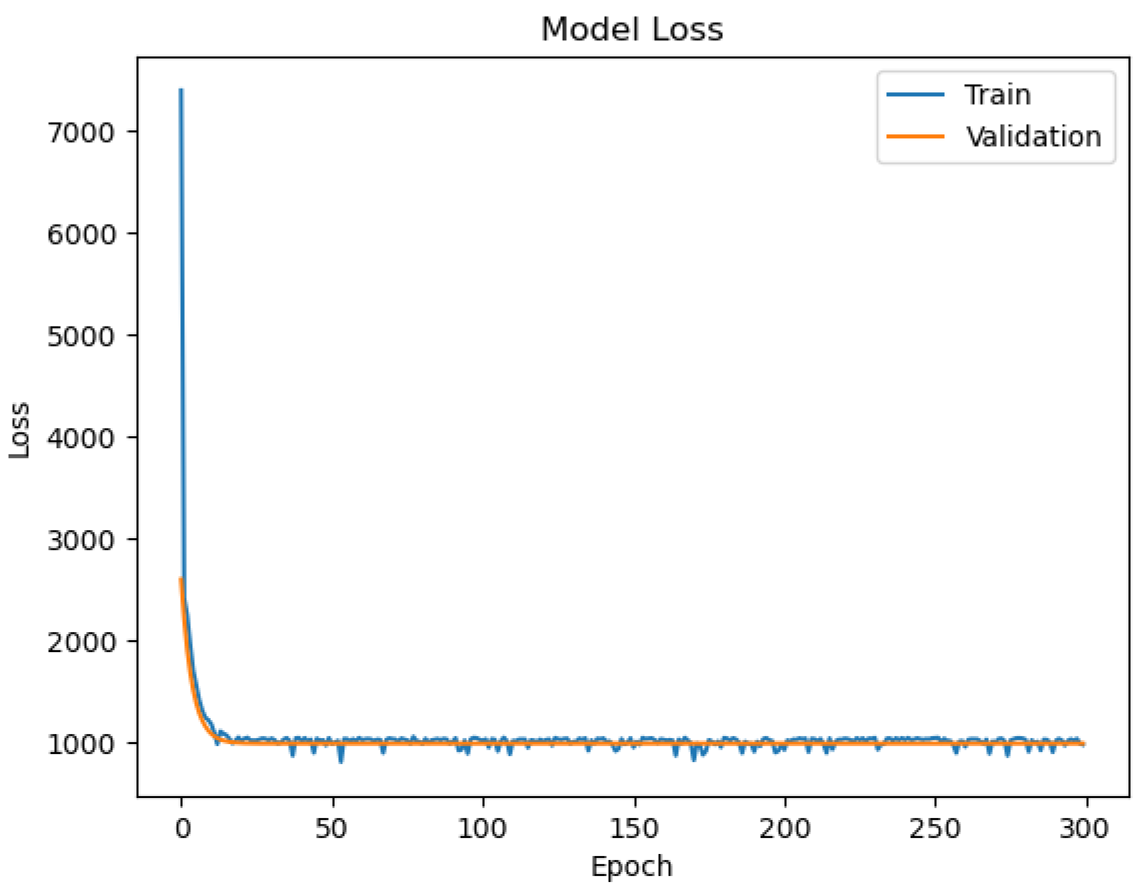
In [ ]:
```
data = np.loadtxt('housing_prices.txt', delimiter=',', skiprows=1)

X = data[:,0]
y = data[:,1]

#define keras model
model = Sequential()

model.add(Dense(2,input_dim=1,activation='relu'))
model.add(Dense(1))

#compile the keras model
opt = optimizers.SGD(learning_rate=0.001)
mse = tf.keras.losses.MeanSquaredError(
    reduction=tf.keras.losses.Reduction.SUM)
model.compile(loss=mse, optimizer=opt)
```

```
WARNING:tensorflow:From c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\sit
e-packages\keras\src\backend.py:873: The name tf.get_default_graph is deprecated. Ple
ase use tf.compat.v1.get_default_graph instead.
```

In [ ]:
```
#Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=

#fit the keras model on the dataset (CPU)
fit_data = model.fit(X_train,y_train,epochs=300, validation_data=(X_train, y_train), v
model.summary()
```

```
WARNING:tensorflow:From c:\Users\MorgadoBruno\AppData\Local\anaconda3\envs\ML\lib\sit
e-packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue is d
eprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 4

 dense_1 (Dense)             (None, 1)                 3

=================================================================
Total params: 7 (28.00 Byte)
Trainable params: 7 (28.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
3/3 [==============================] - 0s 4ms/step
1/1 [==============================] - 0s 56ms/step
```

In [ ]:
```
plt.plot(fit_data.history['loss'])
plt.plot(fit_data.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```



In [ ]:
```
prediction = model.predict([16.5])
print("the neural network model predicts a the price of a house in a city 165 000 peop
```

```
1/1 [==============================] - 0s 40ms/step
the neural network model predicts a the price of a house in a city 165 000 people to
be: $ 61510
1/1 [==============================] - 0s 40ms/step
the neural network model predicts a the price of a house in a city 165 000 people to
be: $ 61510
```

In [ ]:
```
# OLS regression comparison
X = data[:,0]
const = np.ones([np.shape(X)[0], 1])
X_1d = np.append(const, X)
X = np.reshape(X_1d, (np.shape(X)[0], 2), 'F')
beta_analytic = np.linalg.inv(X.T@X)@X.T@y
```

In [ ]:
```
prediction_OLS = beta_analytic[0] + beta_analytic[1] * 16.5
print("the Linear OLS model predicts a the price of a house in a city 165 000 people t
```

```
the Linear OLS model predicts a the price of a house in a city 165 000 people to be:
$ 157893
```

In this case the neural network is likely not to be the most effective model for this purpose

Appendix:

Random snipets of code used to debug

In [ ]:
```
a = np.arange(13)
batch_size = 3
len_a = np.size(a)
print(len_a)
number_of_batches = int(len_a/batch_size)
print(len_a%batch_size)
i = 0

count = 0
while count < 10:
    i = count%number_of_batches
    if i < number_of_batches-1:
        lower_bound = i*batch_size
        upper_bound = (i+1)*batch_size
        print(a[np.arange(lower_bound,upper_bound)])

    elif len_a%batch_size == 0 and i == number_of_batches-1:
        lower_bound = i*batch_size
        upper_bound = (i+1)*batch_size
        print(a[np.arange(lower_bound,upper_bound)])

    elif len_a%batch_size > 0 and i == number_of_batches-1:
        lower_bound = (i+1)*batch_size
        print(a[np.arange(lower_bound,len_a)])


    count += 1
```

```
13
1
[0 1 2]
[3 4 5]
[6 7 8]
[12]
[0 1 2]
[3 4 5]
[6 7 8]
[12]
[0 1 2]
[3 4 5]
```