Intro to Machine Learning - Assignment 5+6

Maxmillan Ries s3118134, Cristian Rosiu s3742377

October 2020

Contents

1 Learning Vector Quantization

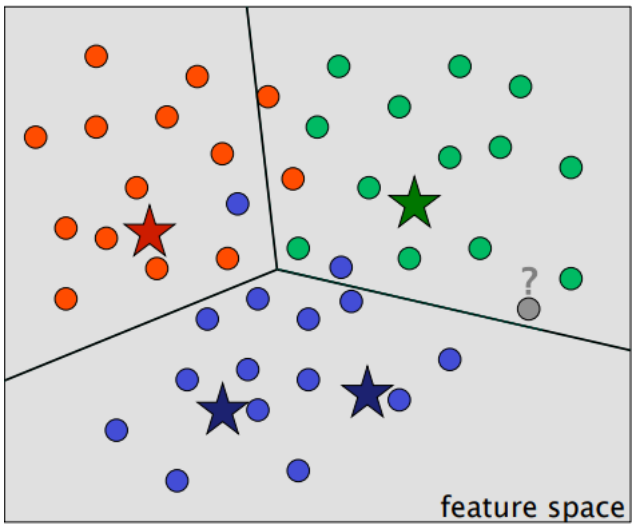
1.1 Introduction

Often times, when referring to the concept of "machine learning", the thought of neural networks appears, almost interchangeably.
However, the reality is very different. While neural networks are very powerful tools which can achieve amazing results towards complex problems, they do not define the concept of machine learning.
In broader terms, machine learning is an algorithm that learns. While the learning kind/method may not be mimicking that of a human, it is still considered learning.

And among the many machine learning algorithms, one particular algorithm is commonly used to classify data by means of prototypes. This algorithm is called "Learning Vector Quantization".
And in the first part of this report, we shall analyze our implementation of the algorithm, and the nature of the algorithm itself.

1.2 What is it?

Learning Vector Quantization (also known as LVQ) is a prototype-based learning method. A single, or several prototypes are defined and used to represent respective classes in a given dataset, with each prototype being shown as a point in the feature space (collection of features related to one or more properties about a studied "object").
The image below consists of an picture in the lecture slides which represents the idea of the result of running Learning Vector Quantization on a set of data.



This method of classification is not known for being the best, but is simple and intuitive to understand and implement. When looking at the image, it is important to note that each star is a prototype and each circle is a data point. And while the colored stars sit well within their respective color-matching data points, the classification is not perfect.
This image in short, represents both the benefits and drawbacks of LVQ. While LVQ can classify data well according to a given metric distance (often Euclidean Distance), the number of prototypes chosen to represent the data is an "ill defined problem", as the blue data could just as easily (and correctly) be represented by a single prototype.

In the first part of this report, we will be running tests on a provided data set, comparing the obtained results when choosing a single prototype per class, versus 2 prototypes per class.

Final code can be found in the Appendix.

1.3 Example of Single vs Dual Prototypes per class

The basic and underlying mechanism behind Learning Vector Quantization consists of updating, for each data point, the winning prototype based on the Euclidean Distance and the learning rate.
Unlike basic vector quantization however, LVQ additionally moves all non-winning prototypes away from the mismatched data points. The learning aspect which is done on top of basic vector quantization, consists of repeating this process numerous times, until the algorithm converges.

After each epoch (iteration over the **randomized** data), the Training/Learning Error is computed by assessing the number of misclassified numbers and deriving it's percentage.

Figure 1: Error curve for K=1 and learning_rate = 0.002

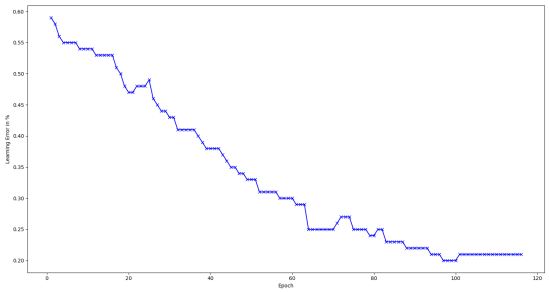
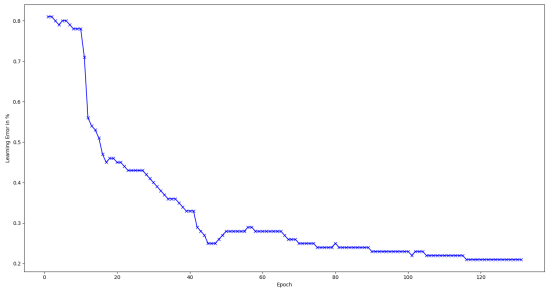


Figure 2: Error curve for K=1 and learning_rate = 0.002



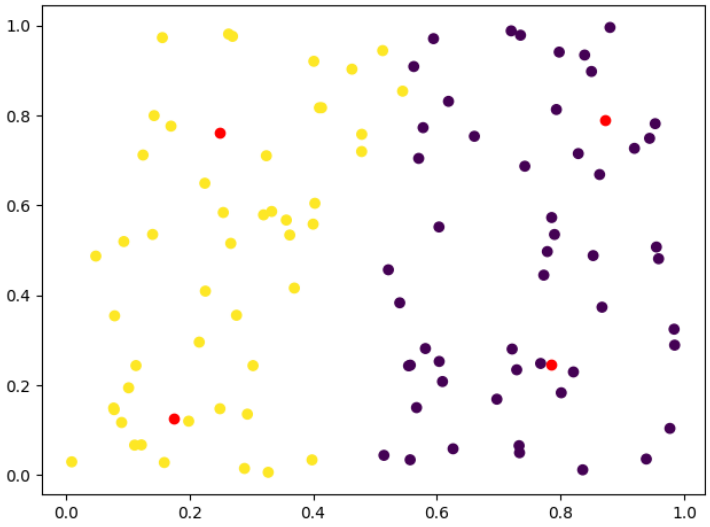
Both graphs representing the training rate look similar. At first, within

the first 10 epoches, the number of mismatched cases decreases sharply as the number the prototypes are ushered towards their respective data with large increments (due to the larger distance between the randomly chosen points).
Eventually, as the prototypes are placed within their respective data points clusters, the error percentage plateaus, eventually reaching a stable continuous state, at around a 20% training error.

It should be noted that the previous statement only applies in theory. In practice, as it is in our code, as the data is randomly presented for each epoch, things never flatten perfectly. As such, a few jumps periodically occur, something which was taken into account by storing the last few data error rates and using them to find a good plateau to stop the training.

1.4 Results of LVQ1

Figure 3: Clusters and their prototypes after running LVQ1 with K=2 and learning_rate=0.002



In the graph above, the red data points represent the final positions of the prototypes. As indicated in the assignment, only case (b) is presented, in which 2 prototypes are assigned per class.
In this graph, we can see that the classification was run "well". The "wellness" of the algorithm's training is defined by comparing it to the kind of clustering we have previously run in assignment 3+4, showing that a spread of data as provided, is split into 4 sections, each holding a prototype in the center.
Moreover, the black vs yellow (white) data sets indeed only hold two prototypes

in them, both being split between the top and the bottom of the respective data points of the class.

1.5 Discussion and Bonus Question

Overall, something to note is that most Training Error graphs obtained were not as perfect as the ones displayed. While the shape of the graph was similar, due to the randomization of the data, the graph was a lot less smooth, making it more difficult to explain the pattern which it represented.

Additionally, given that this algorithm needed to be run for both 1 and 2 prototypes per class, it is only right to discuss and analyze the difference between the two obtained clustering results. As we can clearly see, the prototypes in

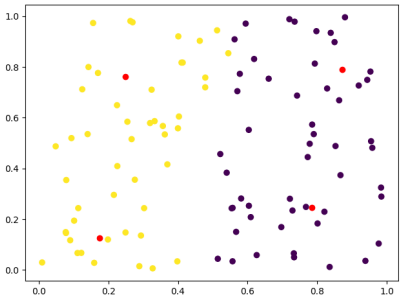


Figure 4: K = 2 and learning_rate = 0.002

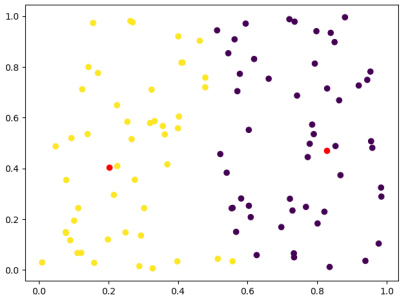


Figure 5: K = 1 and learning_rate = 0.002

both cases are placed in their respective "data center". The graph of (b) shows, as described above, that each class is split into 2 sections, each holding a prototype in it's center, while case (a) shows a similar result to running classic vector quantization or other non-machine learning algorithms.

1.5.1 Bonus Question: 3 and 4 prototypes per cluster

When looking at plotted training error per epoch for 3 and 4 prototypes, we can immediately see that the learning rate error converges similarly to the one where we used 1 and 2 prototypes (i.e. converges towards 0.2)

When observing the plotted 2D data and the prototypes, a clear difference can be observed compared to the use of only 1 and 2 prototypes. While the classification of the data between 1 and 2 prototypes was similar, with 3 and 4 prototypes we usually got final graphs that looked a lot more like the original data and labels.

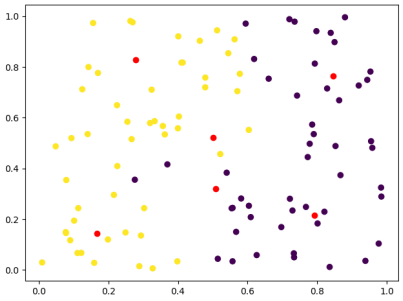


Figure 6: K = 3 and learning_rate = 0.002

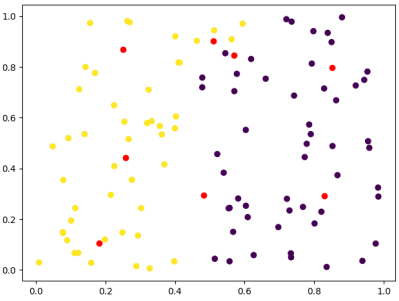


Figure 7: K = 4 and learning_rate = 0.002

Moreover, in many of the runs done to see a pattern across the randomization of the data, the data was no longer split 50/50. Rather than having 2 classes with roughly the same amount of data points, with 3 and 4 prototypes, one class seemed to consistently have more labelled data points than the other.

As a final point to the classification using more than 2 prototypes, seems to be more suitable for the given data-set. It seems that, having more prototypes when the original clusters are somewhat intertwined, helps with the prediction of labels that might be harder to find using 2 or less prototypes.

2 Linear Regression

Linear Regression, in short, finds a linear function (of the form y = mx + b) that, with a minimized error, predicts and models a dependent variable (y) as a function of an independent variable (x).

In the case of this assignment, the relationship between x and y is simplified from the standard linear expression to the form:

```
y = w * x
```

In our case, the given data x has $N = 25$ dimensions ($x \in R^{25}$), and as such, multiple linear regression will be performed.

As per the guidance provided in the assignment, the linear regression process will be split into two stages, training and testing. In the first training stage, a given data set consisting of matching x and y values will be used to infer the vector w's values.
The process will be repeated for different subsets of the data, as to observe, how the number of training samples affects the relationship estimated between the

dependent and independent variable.
The following statements are an extract from the slides:

$$\begin{aligned} w_1 x_1^1 + w_2 x_2^1 \dots w_N x_N^1 &= 0 \\ w_1 x_1^2 + w_2 x_2^2 \dots w_N x_N^2 &= 0 \\ \dots \quad \dots \quad \dots \quad &\dots \\ w_1 x_1^P + w_2 x_2^P \dots w_N x_N^P &= 0 \end{aligned}$$

P equations for N unknowns

$P<N$: in general no unique solution

$P>N$: in general no solution $E=0$ but $\nabla_w E = 0$ is possible

where P is the number of known equations, and N the number of unknowns to be found.

In our case, the value of P ranges from the values of 30 to 500, allowing us to immediately know and determine that there is no singular solution which equates the Mean Square Error to 0. There is a way however of equating it's derivative relative to w to 0.

Final code can be found in the Appendix.

2.1 Moore-Penrose Inverse

This process is done using the Moore-Penrose pseudoinverse. Luckily for us, the Numpy library of Python has the function *pinv*, which allows us to calculate the inverse of the a matrix, simplifying the coding process.
The function to calculate the the values of w based on a given set of x's and y's using the Moore-Penrose pseudoinverse is the following:

$$\rightarrow \quad \overset{N\text{-}dim.}{\mathbf{w}^*} = \underbrace{[\overset{NxN}{X^\top X}]^{-1} \overset{NxP}{X^\top}}_{\text{pseudo-inverse of the rectangular matrix } X} \overset{P\text{-}dim.}{Y}$$

where X is the set of P values of x, Y is the set P matching values of y and the "...]$^{-1}$" represents the inverse of a matrix.

Using this equation, the vector w can be obtained for a given training set, and this same vector can then be used to evaluate the relationship between a given test set.

2.2 Mean Squared Error

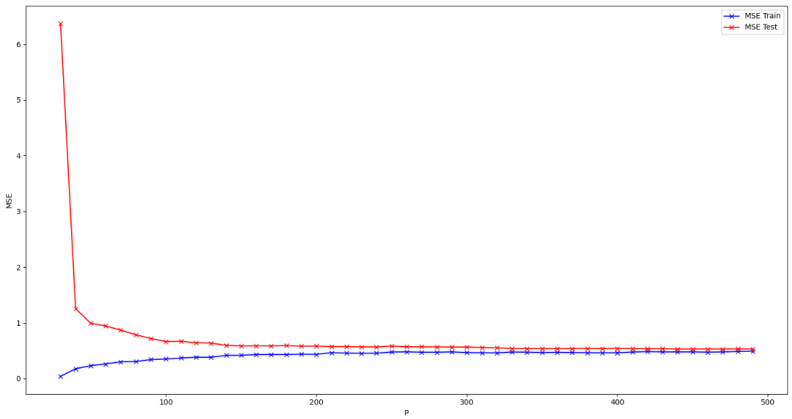
The Mean Squared Error is an error value which evaluates the the difference between an expected value and an actual value. It is calculated using the following expression:

$$E_{train} = \frac{1}{2P} \sum_{\mu=1}^{P} \left(\mathbf{w}^* \cdot \mathbf{x}^{\mu}_{train} - y^{\mu}_{train} \right)^2 .$$

where 1/2P represents the normalization of of the data, as we aim to compare the errors for different values of P.

2.3 Results

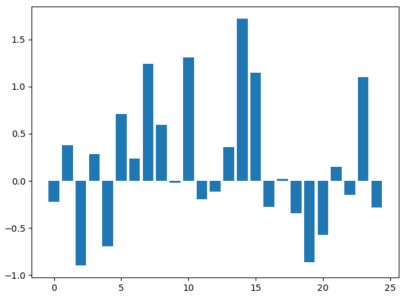
When wrapping it all together in our algorithm and printing the appropriate data, we made sure to create a graph, presenting the difference between the MSE values for Etrain and Etest for different values of P.



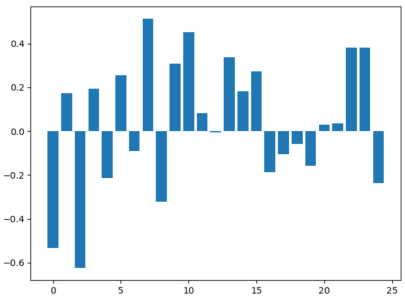
In blue we see the Mean Squared Error for Etrain and in red, the MSE for Etest.

Comparing both errors, we can already see that the values of w obtained matches the testing data very well. For lower values of P, a slight discrepancy can be noticed between both curves, one which does disappear almost entirely for values P=300 and above.
The difference between the respective values of P progressively decreases over time, as an obvious correlation between the number of samples and the accuracy of the vector w is known to exist.
When the training set is very small, it is easy to fit a linear curve such that the error margin itself is close to 0, as described in our subsection above. When those values are applied to the testing set however, the MSE is very large (around 6.4-6.5). This is simply a result of a poorly fitted line being applied to 500 data points.
It is with that same logic that the curves match for high values of P.

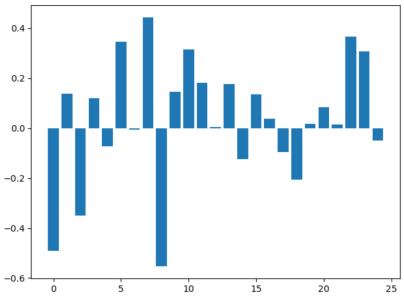
Overall, since a good fit can be identified by training w and validating a minimal change in MSE, we can conclude that our algorithm has produced a good model for the given data.



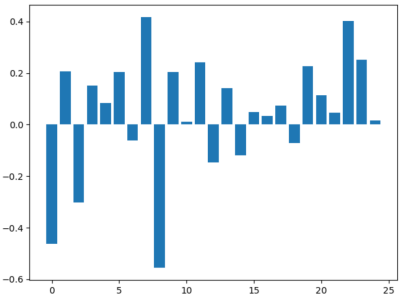
(a) P = 30



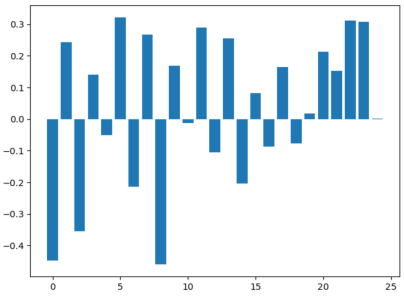
(b) P = 40



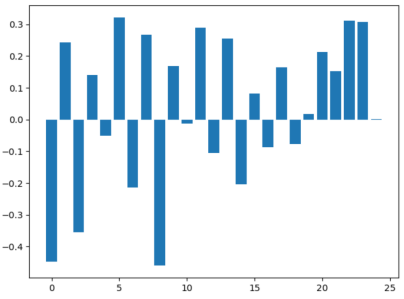
(c) P = 50



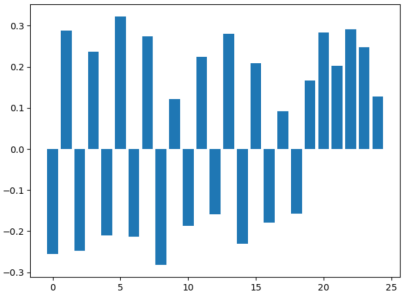
(d) P = 75



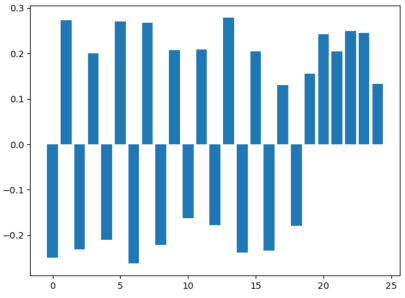
(e) P = 100



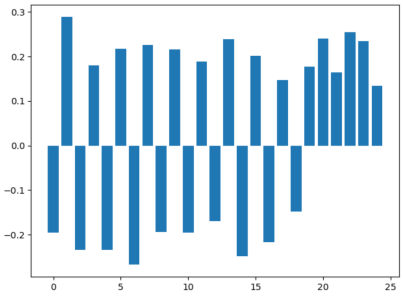
(f) P = 200



(g) P = 300



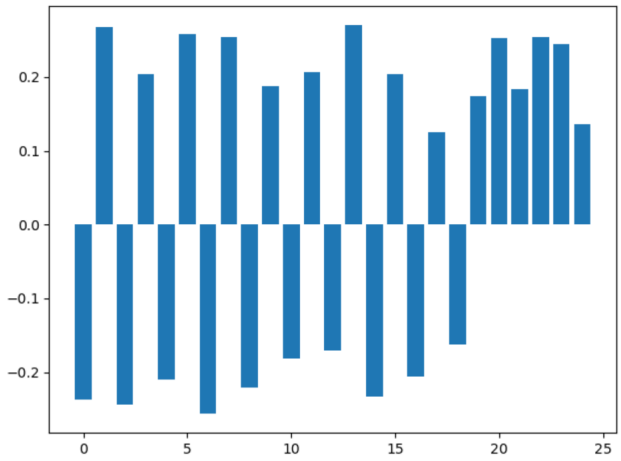
(h) P = 400



(i) P = 500

For the values of P ranging from 30 to 200, we can see a sharper change in value between each iteration displayed. Initially (at P = 30), the graph shows values of high positive magnitude, before the negative values seem to become more pronounced (P 40 - 200).
From P = 200 onwards however, the pattern changes, almost as if stabilizing due to the increased training data provided. Instead of larger changes within several iterations, the values of w seem to remain and oscillate around fixed set of values.

As a guess on how to obtain the original values of w, we took the average of all values of w for a range of values of P between 300 and 500, resulting in the following graph below:



As such, our guess of the original values hold to be the following, or the values obtained after rounding them to the nearest 0.5.

As a conclusion, we have seen throughout our investigation process the impact the amount of training data has on the slope of the linear line created by the regression algorithm.
A small number of training data values leads to a very low error on the data it was trained on as the line can be perfectly fit, but leads to a very large error on any data set used beyond the training one.
For increasing larger number of training points, with the condition that P > N, it is impossible to get the Mean Squared Error to 0, although it is possible to get the change is MSE to be minimal.

3 Appendix

3.1 Code

3.1.1 LVQ1

```python
def euclidean_distance(a, b):
    return np.linalg.norm(a - b)


class LVQ1:
    def __init__(self, n_classes, learning_rate=0.1, prot_per_class=1):
        self.n_classes = n_classes
        self.learning_rate = learning_rate
        self.prot_per_class = prot_per_class
        self.__trained = False

    # Randomly initializes prototypes.
    def __random_init(self, X, y):
        prototypes = []
        prototype_labels = []
        labels = np.unique(y)

        for i in range(self.n_classes):
            for j in range(self.prot_per_class):
                prototypes.append(X[np.random.choice(X.shape[0])])
                prototype_labels.append(labels[i])

        self.prototypes = np.array(prototypes)
        self.prototype_labels = np.array(prototype_labels)

    def __shuffle_data(self, data, labels):
        randomized_data = list(zip(data.tolist(), labels.tolist()))
        np.random.shuffle(randomized_data)
        randomized_data = list(zip(*randomized_data))

        return np.array(randomized_data[0]), np.array(randomized_data[1])

    def __nearest_neighbour(self, point):
        winner = (0, sys.maxsize)
        for index, prototype in enumerate(self.prototypes):
            distance = euclidean_distance(prototype, point)
            if distance < winner[1]:
                winner = (index, distance)
        return winner[0]

```

```
41      def __update_prototype(self, point, point_index,
        ↪  prototype_index, labels):
42          sign = 1 if (self.prototype_labels[prototype_index] ==
            ↪  labels[point_index]) else -1
43          self.prototypes[prototype_index] =
            ↪  self.prototypes[prototype_index] + self.learning_rate *
            ↪  sign * (
44                  self.prototypes[prototype_index] - point)
45
46      def __calculate_error(self, X, y):
47          tp_sum = 0
48          y_pred = self.predict(X)
49          for i, label in enumerate(y_pred):
50              if label == y[i]:
51                  tp_sum += 1
52
53          return tp_sum / 100
54
55      def __plot_epoch(self, X, y):
56          plt.scatter(X[:, 0], X[:, 1], c=y.tolist())
57          plt.scatter(self.prototypes[:, 0], self.prototypes[:, 1],
            ↪  c='r')
58          plt.show()
59
60      def __plot_error(self, epochs, errors):
61          K = range(1, epochs + 1)
62          plt.figure(figsize=(15, 8))
63          plt.plot(K, errors, 'bx-')
64          plt.xlabel('Epoch')
65          plt.ylabel('Learning Error in %')
66          plt.show()
67
68      def train(self, X, y, epochs=10, plot_error=False,
        ↪  plot_epoch=False, stop_threshold=10):
69          self.__random_init(X, y)
70          errors = []
71          prev_error = 0
72          duplicates = 0
73          for epoch in range(epochs):
74              if duplicates >= stop_threshold:
75                  epochs = epoch
76                  break
77              points, targets = self.__shuffle_data(X, y)
78              for index, point in enumerate(points):
79                  winner_index = self.__nearest_neighbour(point)
80                  self.__update_prototype(point, index, winner_index,
                    ↪  targets)
81              errors.append(self.__calculate_error(X, y))
82              if epoch == 0:
```

```
                prev_error = errors[0]
            else:
                current = errors[len(errors) - 1]
                if prev_error == current:
                    duplicates += 1
                else:
                    duplicates = 0
                prev_error = current
        if plot_epoch:
            self.__plot_epoch(X, y)
        self.__trained = True
        if plot_error:
            self.__plot_error(epochs, errors)

    def predict(self, samples):
        predicted_labels = []
        for point in samples:

            ↪ predicted_labels.append(self.prototype_labels[self.__nearest_neighbour(point)])
        return predicted_labels
```

3.1.2 Linear Regression

```
class LinearRegression:
    def __init__(self):
        self.weights = np.array([])

    def MSE(self, X, y):
        sum = 0
        for i, point in enumerate(X):
            sum = sum + pow(np.dot(self.weights, point) - y[i], 2)

        return sum / (2 * len(X))

    def train(self, X, y):
        p_inverse = np.linalg.pinv(X.T.dot(X))
        self.weights = p_inverse.dot(X.T.dot(y))
```

3.1.3 LVQ Data-set

```
from slearning import LVQ1
import pandas
import numpy as np
import matplotlib.pyplot as plt

X = pandas.read_csv('datasets/lvqdata.csv')
```

```
X = X.to_numpy()

y = np.array([1 if i < 50 else 2 for i in range(X.shape[0])])

# Plot error
prototypes = [1, 2]
for p in prototypes:
    lvq = LVQ1(n_classes=2, learning_rate=0.002, prot_per_class=p)
    lvq.train(X=X, y=y, epochs=200, plot_error=True,
    ↪  stop_threshold=15)

# Plot graph
lvq = LVQ1(n_classes=2, learning_rate=0.002, prot_per_class=2)
lvq.train(X=X, y=y, epochs=200, plot_epoch=True, stop_threshold=40)
plt.scatter(X[:, 0], X[:, 1], c=lvq.predict(X))
plt.scatter(lvq.prototypes[:, 0], lvq.prototypes[:, 1], c='red')
plt.show()
```

3.1.4 Linear Regression Data-set

```
from slearning import LinearRegression
import numpy as np
import matplotlib.pyplot as plt


def plot_error(X, y, X_test, y_test):
    P = range(30, 500, 10)
    train_error = []
    test_error = []
    for size in P:
        lr = LinearRegression()
        lr.train(X[:size], y[:size])
        train_error.append(lr.MSE(X, y))
        test_error.append(lr.MSE(X_test, y_test))

    plt.figure(figsize=(15, 8))
    plt.plot(P, train_error, 'bx-')
    plt.plot(P, test_error, 'bx-', color='r')
    plt.legend(('MSE Train', 'MSE Test'))
    plt.xlabel('P')
    plt.ylabel('MSE')
    plt.show()


def plot_weights(X, y, X_test, y_test):
    P = [30, 40, 50, 75, 100, 200, 300, 400, 500]

```

```
    for size in P:
        lr = LinearRegression()
        lr.train(X[:size], y[:size])

        x_values = range(25)
        plt.bar(x_values, list(lr.weights))
        plt.show()


if __name__ == '__main__':
    X = np.genfromtxt('datasets/xtrain.csv', delimiter=',', skip_header=True)
    y = np.genfromtxt('datasets/ytrain.csv', delimiter=',', skip_header=True)

    X_test = np.genfromtxt('datasets/xtest.csv', delimiter=',', skip_header=True)
    y_test = np.genfromtxt('datasets/ytest.csv', delimiter=',', skip_header=True)

    plot_error(X, y, X_test, y_test)
    plot_weights(X, y, X_test, y_test)
```