wine

October 15, 2024

[78]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt


df = pd.read_csv('wine.csv', header=None)
df.columns = ['Type', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash',\
     'Magnesium', 'Total phenols', 'Flavanoids', 'Nonflavanoid␣
 ↪phenols','Proanthocyanins', 'Color intensity', 'Hue', 'OD280/OD315 of␣
 ↪diluted wines', 'Proline']

# df.head(5)
filtered_df = df[df['Type'] != 3]
print(filtered_df.groupby('Type').size())
```

```
Type
1    59
2    71
dtype: int64
```

csv column pandas drop 3

[79]:
```
y = filtered_df.iloc[:,0]
X = filtered_df.iloc[:,1:]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state=3)
```

Mission Accomplished: 0.3

[80]:
```
class Perceptron():

    def __init__(self, n_feature = 13, learning_rate = 1e-3, epochs = 100,␣
 ↪tolerance = None, patience = 10):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.W = np.random.random(n_feature + 1) * 0.5
```

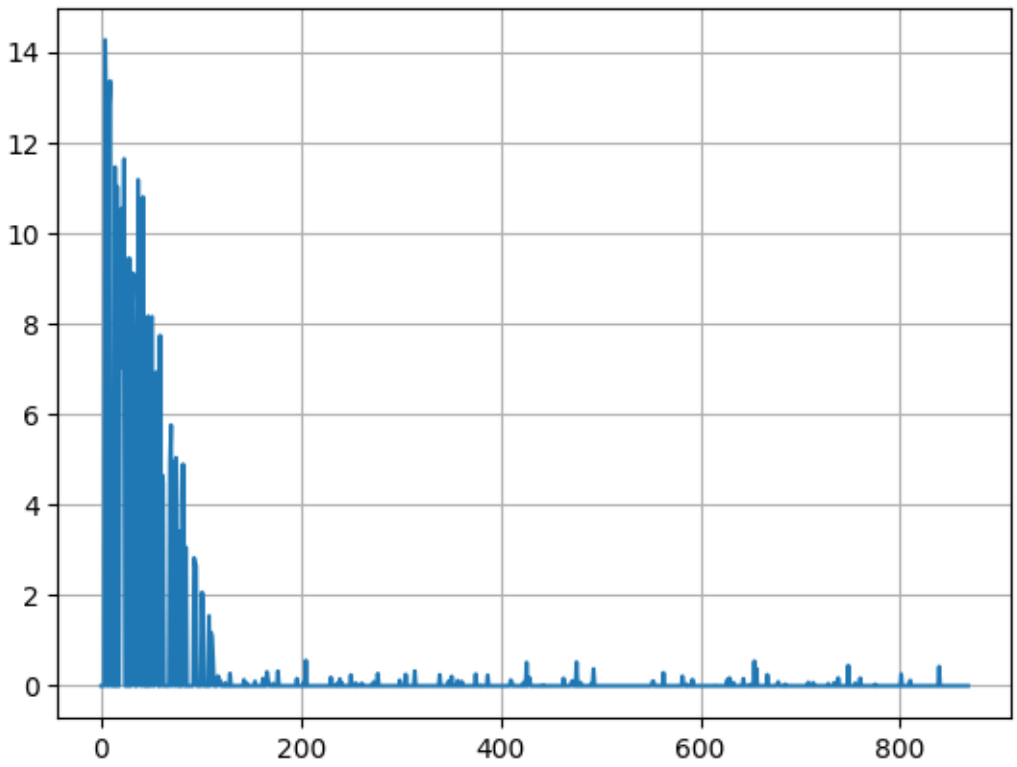
```
        self.W = np.random.uniform(0.01, 0.01, n_feature + 1)
        self.loss = []
        self.best_loss = np.inf

        self.tol = tolerance
        self.patience = patience


    def _loss(self, y, y_pred):
        return - y_pred * y if y_pred * y < 0 else 0

    def _gradient(self, x_bar, y, y_pred):
        return -y * x_bar if y_pred * y < 0 else 0

    def _preprocess_data(self, X):
        m, n = X.shape
        X_ = np.empty([m, n+1])
        X_[:, 0] = 1
        X_[:, 1:] = X
        return X_

    def _map_y(self, y):
        mapper = lambda y: -1 if y == 1 else 1
        return np.array([mapper(yi) for yi in y])

    def _predict(self, X):
        return X @ self.W

    def SGD(self, X_train, y):
        X_train_bar = self._preprocess_data(X_train)
        # breakout = False
        y = self._map_y(y)
        epoch_no_improve = 0
        # self.loss.append(self._loss(y, self._predict(X_train_bar)))

        for epoch in range(self.epochs):
            shuffle_index = np.random.permutation(X_train_bar.shape[0])
            X_train_bar = X_train_bar[shuffle_index]
            y = y[shuffle_index]

            for i in range(X_train_bar.shape[0]):
                x_bar = X_train_bar[i]
                y_pred = self._predict(x_bar)
                loss = self._loss(y[i], y_pred)
                self.loss.append(loss)

                # A simple grad desc without considering earlystopping
```

```
                grad = self._gradient(x_bar, y[i], y_pred)
                self.W -= self.learning_rate * grad

                # ----------------------------
                #      end of grad desc
                #         one sample
                # ----------------------------

                # update-based early stopping
                if self.tol is not None:
                    if loss < self.best_loss - self.tol and loss != 0:
                        self.best_loss = loss
                        epoch_no_improve = 0
                    elif np.abs(loss - self.best_loss) < self.tol:
                        epoch_no_improve += 1
                        if epoch_no_improve == self.patience:
                            print(f'Early stopping at epoch {epoch}')
                            return

                # Why use another variable called break? Let's first try using␣
↪return.

    def BGD(self, X_train, y):
        X_train_bar = self._preprocess_data(X_train)
        y = self._map_y(y)
        epoch_no_improve = 0


        for epoch in range(self.epochs):
            shuffle_index = np.random.permutation(X_train_bar.shape[0])
            X_train_bar = X_train_bar[shuffle_index]
            y = y[shuffle_index]

            y_pred = self._predict(X_train_bar)
            loss = self.batch_loss(y, y_pred)
            scalar_loss = np.sum(loss) / X_train_bar.shape[0]
            self.loss.append(scalar_loss) # we sum the loss of all samples,␣
↪into a scalar

            grad = self.batch_gradient(X_train_bar, y, y_pred)
            self.W -= self.learning_rate * grad

            if self.tol is not None:
                if scalar_loss < self.best_loss - self.tol and scalar_loss != 0:
                    self.best_loss = scalar_loss
                    epoch_no_improve = 0
                elif np.abs(scalar_loss - self.best_loss) < self.tol:
```

```
                    epoch_no_improve += 1
                    if epoch_no_improve == self.patience:
                        print(f'Early stopping at epoch {epoch}')
                        return



    def plot_loss(self):
        plt.plot(self.loss)
        plt.grid()
        plt.show()

    def predict(self, X):
        X_bar = self._preprocess_data(X)
        return np.sign(self._predict(X_bar))

    def batch_loss(self, y, y_pred):
        loss = np.where( y == y_pred, 0, np.abs(y * y_pred))
        return loss

    def batch_gradient(self, x_bar, y, y_pred):
        gradient = np.where((y_pred * y)[:, np.newaxis] < 0, -y[:,np.newaxis] *␣
↪x_bar, 0)
        return np.sum(gradient, axis=0)
```

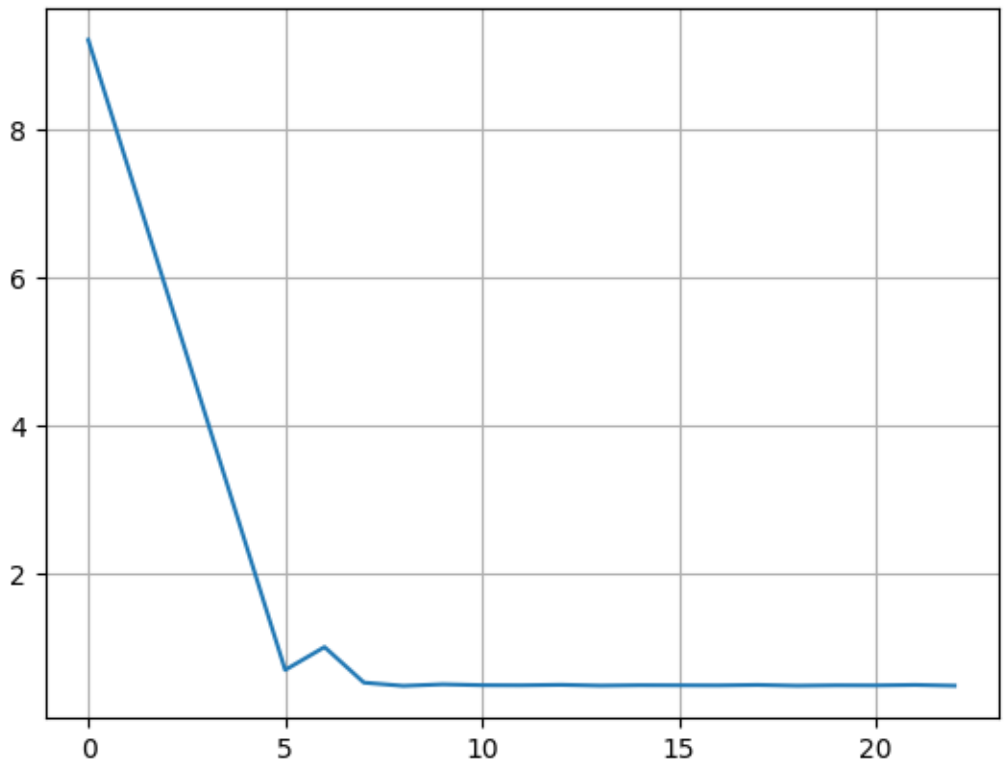
[81]:
```
# we need some conversion between numpy and pandas
X_train = X_train.to_numpy() if isinstance(X_train, pd.DataFrame) else X_train
y_train = y_train.to_numpy() if isinstance(y_train, pd.Series) else y_train

model = Perceptron()
model.learning_rate = 2e-7
model.epochs = 100
model.tol = 1e-2
model.SGD(X_train, y_train)
model.plot_loss()
```

```
Early stopping at epoch 9
```

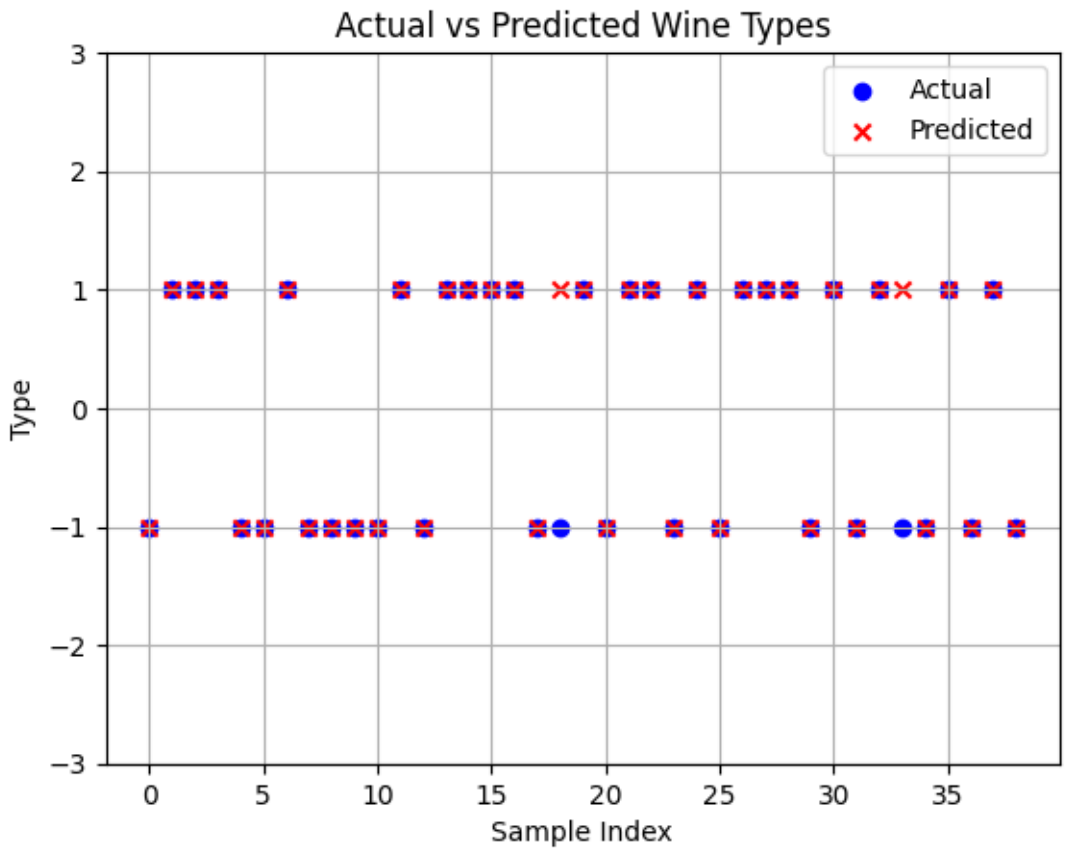


[82]:
```
# Now we do a BGD version

X_train = X_train.to_numpy() if isinstance(X_train, pd.DataFrame) else X_train
y_train = y_train.to_numpy() if isinstance(y_train, pd.Series) else y_train

bgd_model = Perceptron()
bgd_model.learning_rate = 5e-8
bgd_model.epochs = 100
bgd_model.tol = 1e-2
bgd_model.BGD(X_train, y_train)
bgd_model.plot_loss()
```

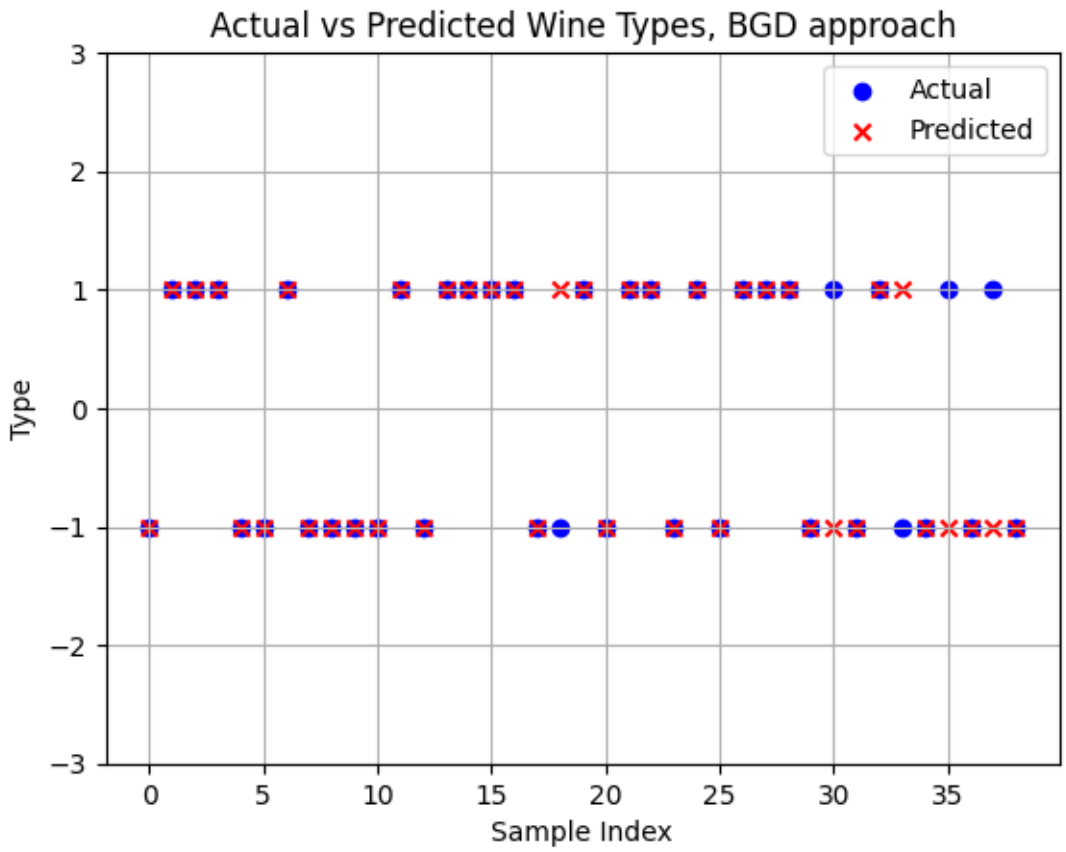
```
Early stopping at epoch 22
```



[83]:
```
mapper = lambda y: -1 if y == 1 else 1
y_mapped = np.array([mapper(yi) for yi in y_test])

# Predict the test set
y_pred = model.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(range(len(y_test)), y_mapped, color='blue', label='Actual')
plt.scatter(range(len(y_test)), y_pred, color='red', label='Predicted',␣
 ↪marker='x')
plt.xlabel('Sample Index')
plt.ylabel('Type')
plt.ylim(-3,3)
plt.title('Actual vs Predicted Wine Types')
plt.legend()
plt.grid(True)
plt.show()
```



[84]:
```
mapper = lambda y: -1 if y == 1 else 1
bgd_y_mapped = np.array([mapper(yi) for yi in y_test])

# Predict the test set
bgd_y_pred = bgd_model.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(range(len(y_test)), bgd_y_mapped, color='blue', label='Actual')
plt.scatter(range(len(y_test)), bgd_y_pred, color='red', label='Predicted',␣
 ↪marker='x')
plt.xlabel('Sample Index')
plt.ylabel('Type')
plt.ylim(-3,3)
plt.title('Actual vs Predicted Wine Types, BGD approach')
plt.legend()
plt.grid(True)
plt.show()
```



metric

[85]:
```
def calculate_accuracy(y_actual, y_pred):

    correct_predictions = 0
    total_predictions = len(y_actual) #  y_actual y_pred

    for i in range(total_predictions):
        if y_actual[i] == y_pred[i]:
            correct_predictions += 1

    return correct_predictions / total_predictions

def calculate_recall(y_actual, y_pred, positive_label=1):
    true_positive = 0
    false_negative = 0

    for i in range(len(y_actual)):
        if y_actual[i] == positive_label:
            if y_pred[i] == positive_label:
```

```
                true_positive += 1
            else:
                false_negative += 1

    recall = true_positive / (true_positive + false_negative)
    # tp + fn
    return recall

def calculate_precision(y_actual, y_pred, positive_label=1):
    true_positive = 0
    false_positive = 0

    for i in range(len(y_actual)):
        if y_pred[i] == positive_label:
            if y_actual[i] == positive_label:
                true_positive += 1
            else:
                false_positive += 1

    precision = true_positive / (true_positive + false_positive)
    #
    #         recall   precision
    #         recall

    return precision

def calculate_f1_score(y_actual, y_pred, positive_label=1):
    precision = calculate_precision(y_actual, y_pred, positive_label)
    recall = calculate_recall(y_actual, y_pred, positive_label)

    f1_score = 2 * precision * recall / (precision + recall)
    #   precision = 1, recall = 1.
    #       f1_score
    return f1_score


accuracy = calculate_accuracy(y_mapped, y_pred)
recall = calculate_recall(y_mapped, y_pred)
precisio = calculate_precision(y_mapped, y_pred)
f1_score = calculate_f1_score(y_mapped, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Recall: {recall}')
print(f'Precision: {precisio}')
print(f'F1 Score: {f1_score}')
```

```
Accuracy: 0.9487179487179487
```

```
Recall: 1.0
Precision: 0.9090909090909091
F1 Score: 0.9523809523809523
```

metric

accuracy recall precision F1 Score recall precision