Lane Moseley

CSC 448: Machine Learning, Fall 2020

Term Project

Table of Contents

Description

This is a multi-part project to build a machine learning library from scratch using Python. The library is called `ML.py`. Included with the library are sample scripts demonstrating the usage of the various components of the library.

Usage

Setup & Requirements

For the `ML.py` library to be useable, a few dependencies must be met. These dependencies are given in `requirements.txt` which is located in the root project directory. Installation is accomplished using the following pip command:

```
    pip3 install -r requirements.txt
```

A Python virtual environment can be used to help simplify dependency and package management across multiple applications. When using a Python virtual environment, dependencies are installed into the virtual environment and not into your system-wide Python installation.

Importing the ML Library

The ML library is implemented in `ML.py`. To use this library in your own code you must first import it. For example, to use the `Perceptron` class and `plot_decision_regions` helper function the following import statement would be used:

```
     from ML import Perceptron, plot_decision_regions
```

k-Nearest Neighbors

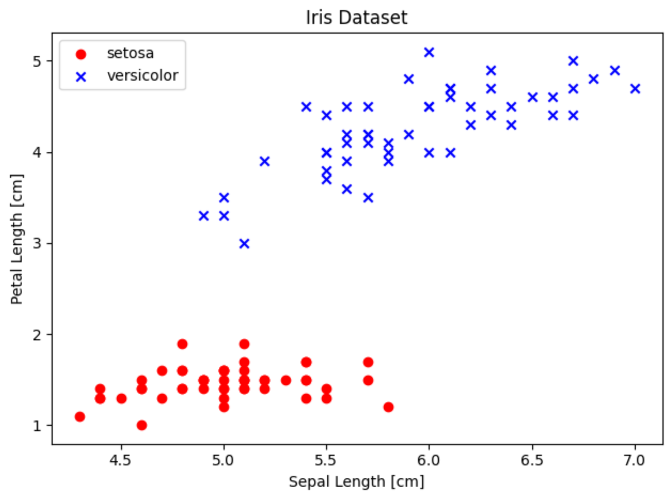
Description

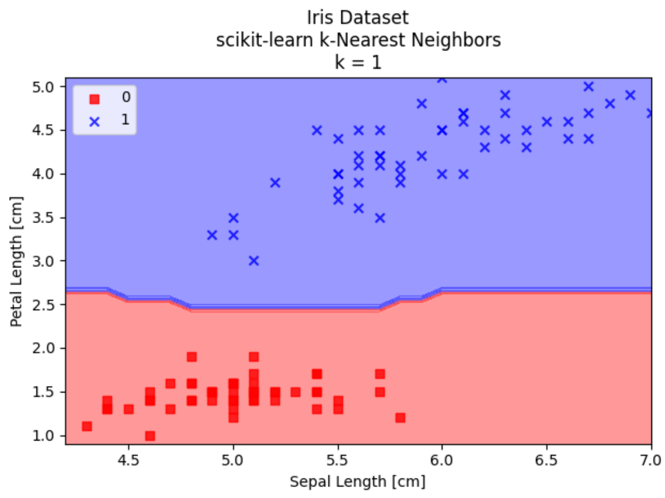
The k-Nearest Neighbors algorithm is a very simple supervised learning algorithm that can be used to classify binary data. The `ML.py` implementation of the k-Nearest Neighbor classifier currently only works for *k=1*, though it could easily be extended to work with higher values of *k*. This means that, at this moment, the input data point is simply assigned the same label as that of the nearest neighboring data point. For higher values of *k*, the classifier would look at the *k* closest neighbors in the feature space and assign the input data point to the class that the majority of its nearest *k* neighbors belong to. Data points can be points in any dimension (i.e., each data point can have any number of features) and Euclidean distance is used to make the comparison. The example below uses the k-Nearest Neighbor classifier with *k=1* to predict whether datapoints from the Iris dataset belong to a given class or not.

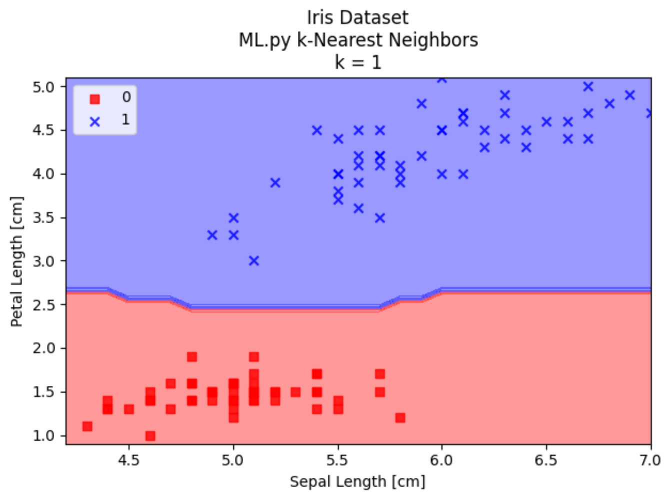
Usage

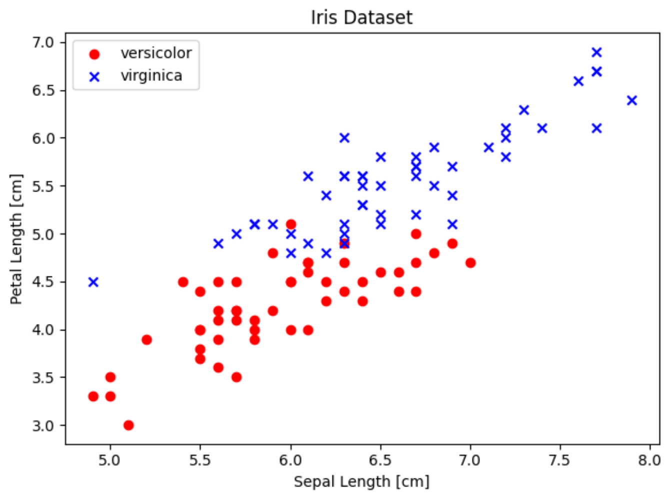
A file called `kNN.py` has been included that demonstrates the usage of the NearestNeighbors class included with the `ML.py` library. The demonstration uses data from the famous Iris Data Set for machine learning. The scikit-learn k-Neighbors Classifier is used to check the accuracy of the `ML.py` k-Nearest Neighbors classifier.

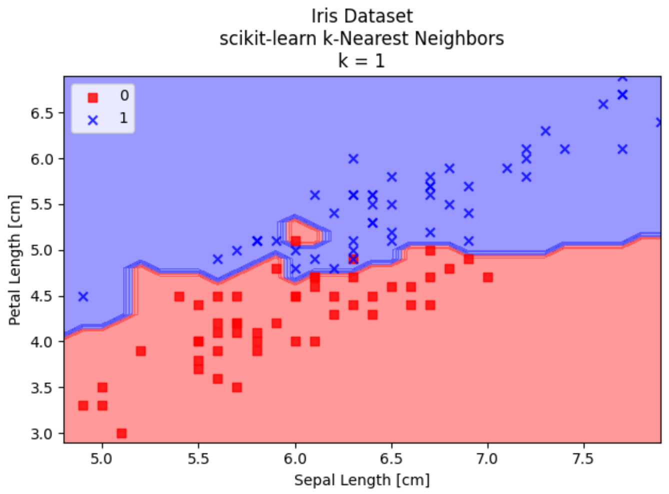
To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 kNN.py` in the root directory of the project. Example screenshots of the expected output are provided below.

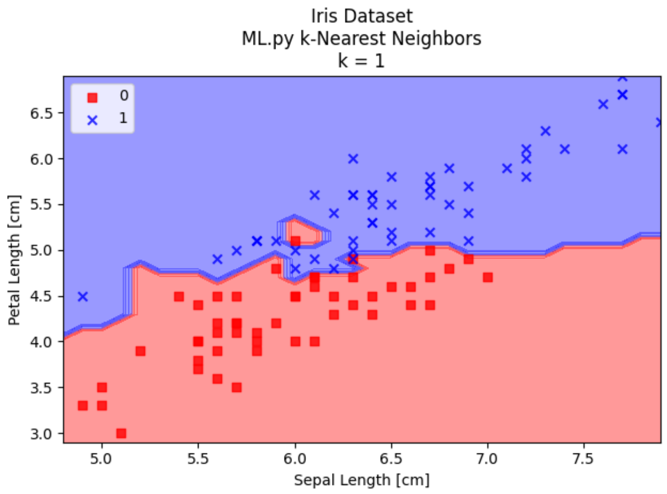












Files & Functions

kNN.py

This file contains code that demonstrates the usage of the NearestNeighbors class.

ML.py

```
class NearestNeighbors(builtins.object)
 |  NearestNeighbors(k=1)
 |
 |  This is the k-nearest neighbors implementation for the ML library.
 |
 |  Methods defined here:
 |
 |  __init__(self, k=1)
 |      Initialize k-nearest neighbors.
 |
 |      Args:
 |          k: number of nearest neighbors to consider
 |
 |  euclidean_distance(self, A, B)
 |      Helper function to get the Euclidean distance between two points.
 |
 |      Args:
 |          A: point A, numpy array
 |          B: point B, numpy array
 |
 |      Returns: Euclidean distance between A and B
 |
 |  fit(self, X, y)
 |      Fit training data.
 |
 |      Args:
 |          X : Training vectors, X.shape : [#samples, #features]
 |          y : Target values, y.shape : [#samples]
 |
 |  predict(self, X)
 |      Return the predicted Y values.
 |
 |      Args:
 |          X_test: X_test : X test vector
 |
 |      Returns:
```

```
 |      Y_pred : Y prediction vector
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Testing

The NearestNeighbors class was tested using the famous Iris Data Set for machine learning. The expected output of `kNN.py` is given above. Part of the above output is the output of the scikit-learn k-Neighbors Classifier, which was used to check the correctness of the `ML.py` k-Nearest Neighbors classifier.

Support Vector Machines

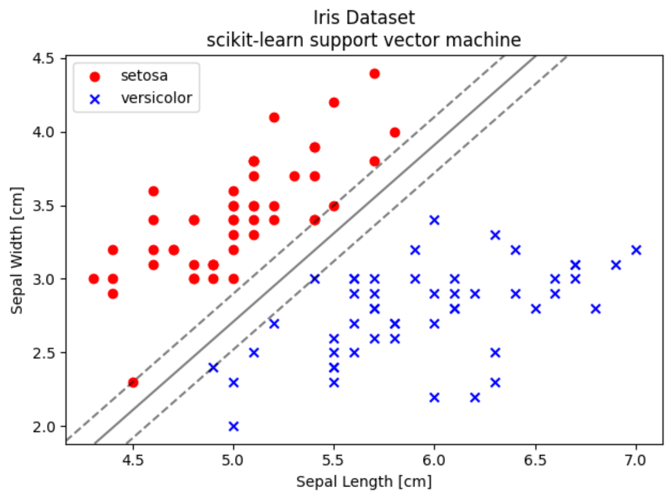
Description

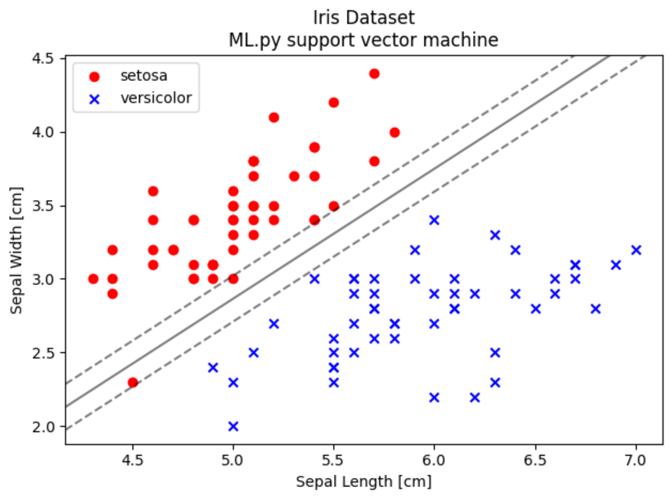
Support Vector Machines (SVMs) are used for learning linear predictors in high-dimensional feature spaces. High dimension feature spaces are challenging to work with because the data is much more complex, however; SVMs can work efficiently even in high-dimensional feature spaces. SVMs typically work by looking at the training data and using gradient descent with a hinge loss function to find a hyperplane that separates the data by the largest possible margin. The SVM implemented in the `ML.py` library is a Hard-SVM, which means that the hyperplane separating the training set does so with the largest possible margin. Note that the tacit requirement here is that the training data is linearly separable. If this requirement is untenable, a Soft-SVM can be used, however; that is beyond the scope of this project.

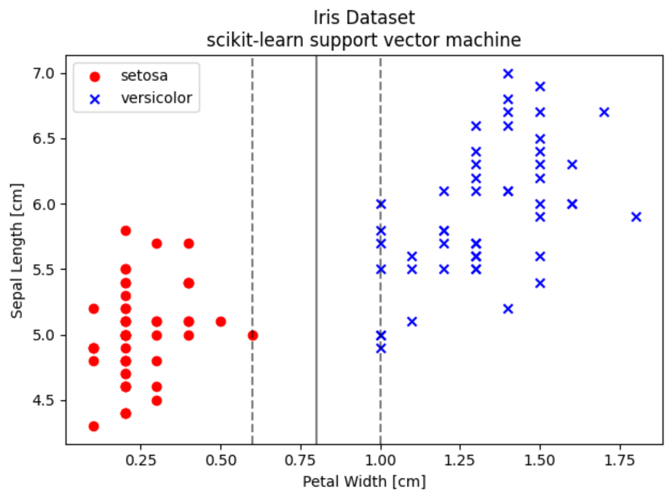
Usage

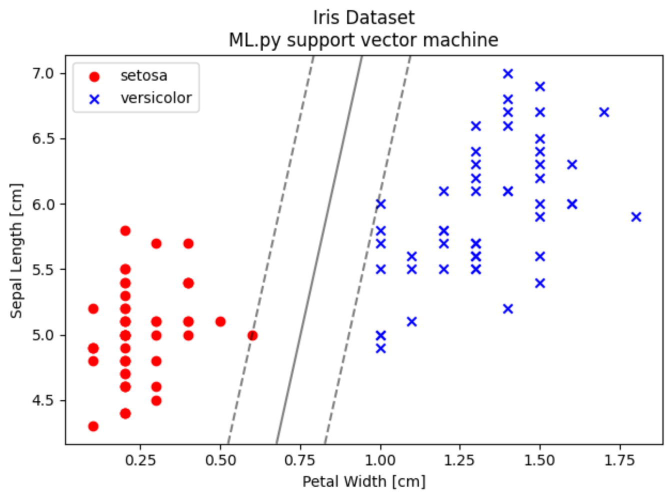
A file called `SVM.py` has been included that demonstrates the usage of the SupportVectorMachine class included with the `ML.py` library. The demonstration uses data from the famous Iris Data Set for machine learning. The scikit-learn Support Vector Classifier is used to check the accuracy of the `ML.py` Support Vector Machine.

To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 SVM.py` in the root directory of the project. Example screenshots of the expected output are provided below.









Files & Functions

SVM.py

This file contains code that demonstrates the usage of the SupportVectorMachine class.

ML.py

```
class SupportVectorMachine(builtins.object)
 |  SupportVectorMachine(learning_rate=0.001, iterations=1000, R=100)
 |
 |  This is the support vector machine implementation for the ML library.
 |
 |  Methods defined here:
 |
 |  __init__(self, learning_rate=0.001, iterations=1000, R=100)
 |      Initialize the support vector machine.
 |
 |      Args:
 |          learning_rate (float, optional): used to scale the weight array.
 |          iterations (int, optional): number learning iterations.
 |          R: regularization parameter, strength of regularization is inversely proportional to R
 |
 |  decision_function(self, X)
 |      Compute decision function w.T * X - b
 |
 |  fit(self, X, y)
 |      Fit training data using hard margin classification. Find weights and
 |      bias that maximize the margin between the two classes of data.
 |
 |      Args:
 |          X : Training vectors, X.shape : [#samples, #features]
 |          y : Target values, y.shape : [#samples]
 |
 |  predict(self, X)
 |      Return the predicted Y values.
 |
 |      Args:
 |          X_test: X_test : X test vector
 |
 |      Returns:
 |          Y_pred : Y prediction vector
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

plot_svc_decision_function(model, ax=None, plot_support=True)
    This is a helper function to plot the decision function for a 2D SVC.

    Author: Jake VanderPlas
    Source: Python Data Science Handbook
    License: MIT
    URL: https://jakevdp.github.io/PythonDataScienceHandbook/05.07-support-vector-machines.html
```

Testing

The SupportVectorMachine class was tested using the famous Iris Data Set for machine learning. The expected output of `SVM.py` is given above. Part of the above output is the output of the scikit-learn Support Vector Classifier, which was used to check the correctness of the `ML.py` Support Vector Machine.

As seen above, a new visualization module was added to facilitate the testing and presentation of the `ML.py` Support Vector Machine. This module displays the decision boundary (the middle-most, solid line) and the support vectors (the two dashed lines on either side of the decision boundary). **The source code for this visualization module was borrowed from the *Python Data Science Handbook* by Jake VanderPlas and is licensed under the MIT license.**

Perceptron

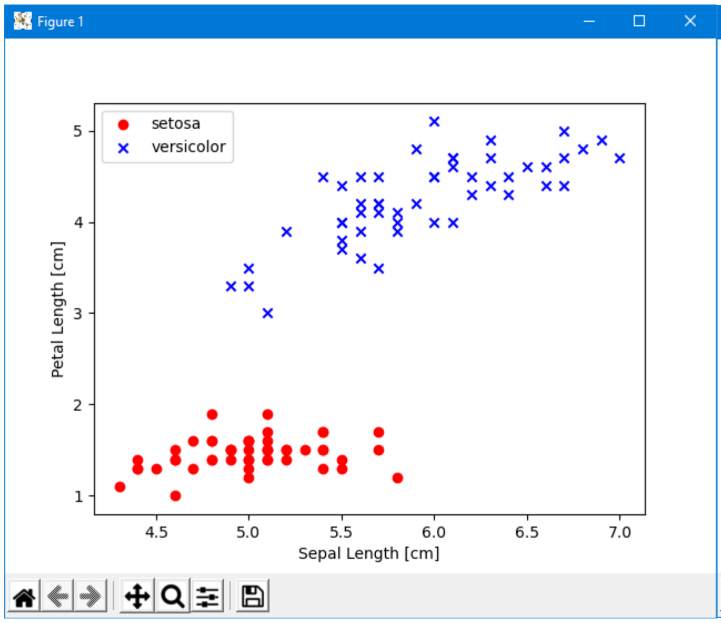
Description

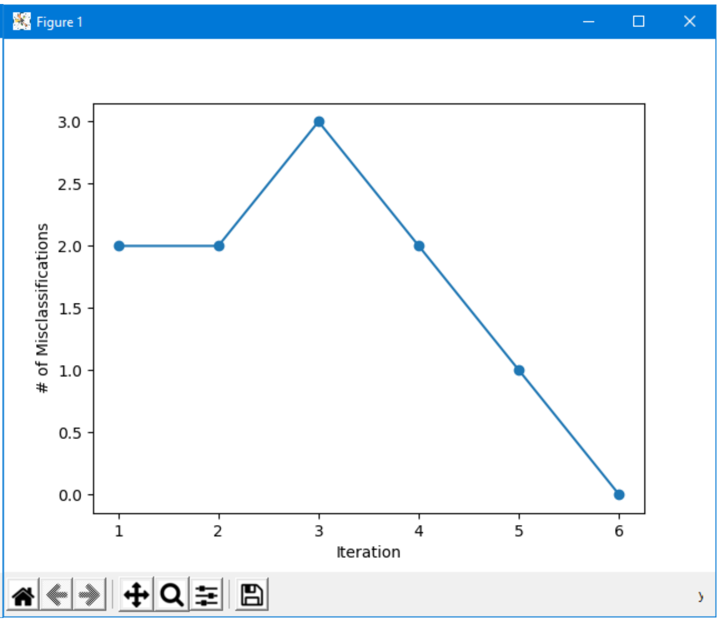
A perceptron is a linear classifier that maps real-valued inputs to a binary output, or label. This means that in order for a perceptron to converge (classify all inputs correctly), the data must be linearly separable. Linearly separable means that it is possible to separate positive and negative examples with a hyperplane. At a high level the perceptron works by iterating over the training data, calculating the predicted values, calculating the error between the predicted values and the expected values, and finally updating the weights and bias in order to improve the error on the next iteration.

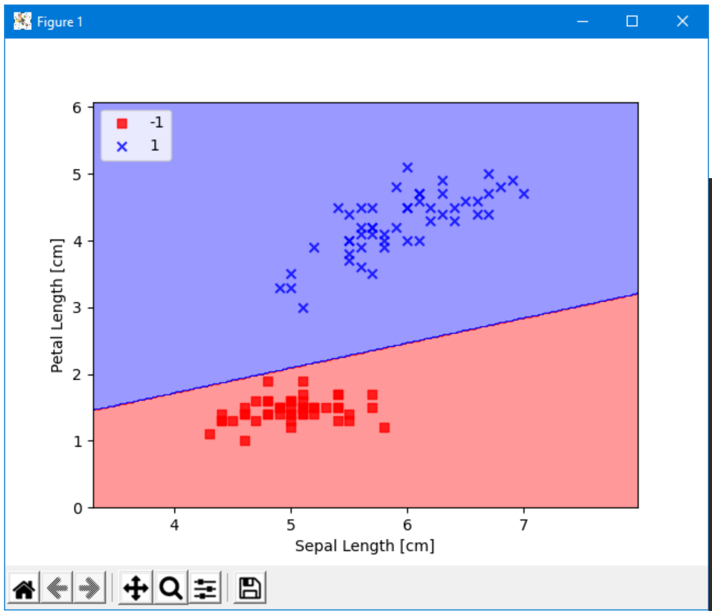
Usage

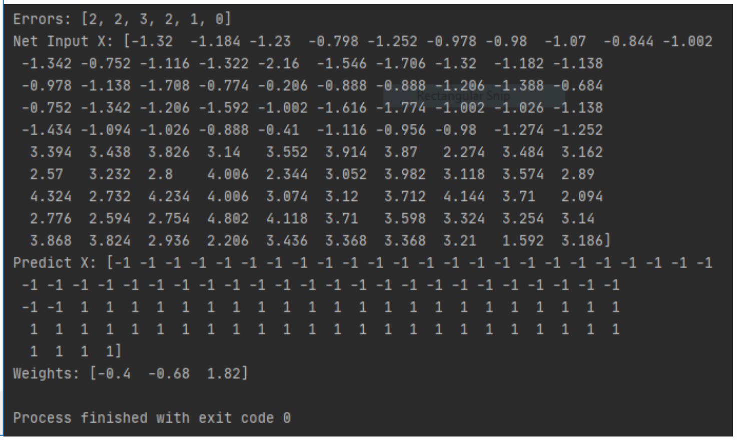
A file called `perceptron.py` has been included that demonstrates the usage of the Perceptron class included with the `ML.py` library. This file also demonstrates the usage of the decision boundary visualization module. The demonstration uses data from the famous Iris Data Set for machine learning.

To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 perceptron.py` in the root directory of the project. The program will get the sepal length and petal length for the Iris-setosa and Iris-versicolor data. It will then show a scatter plot of this data to illustrate that the data is separable. Next, the program will instantiate a perceptron using the ML library and train the perceptron on the sample data. The features used to train the Perceptron are sepal length and petal length. A graph will be displayed that plots the number of misclassifications that occurred during training. Finally, the program will use the `plot_decision_regions` function from the ML library to plot the decision surface. The decision surface shows the partition between the two classes of objects (Iris-setosa and Iris-versicolor). Example screenshots of the expected output are provided below.









```
Errors: [2, 2, 3, 2, 1, 0]
Net Input X: [-1.32  -1.184 -1.23  -0.798 -1.252 -0.978 -0.98  -1.07  -0.844 -1.002
 -1.342 -0.752 -1.116 -1.322 -2.16  -1.546 -1.706 -1.32  -1.182 -1.138
 -0.978 -1.138 -1.708 -0.774 -0.206 -0.888 -0.888 -1.206 -1.388 -0.684
 -0.752 -1.342 -1.206 -1.592 -1.002 -1.616 -1.774 -1.002 -1.026 -1.138
 -1.434 -1.094 -1.026 -0.888 -0.41  -1.116 -0.956 -0.98  -1.274 -1.252
  3.394  3.438  3.826  3.14   3.552  3.914  3.87   2.274  3.484  3.162
  2.57   3.232  2.8    4.006  2.344  3.052  3.982  3.118  3.574  2.89
  4.324  2.732  4.234  4.006  3.074  3.12   3.712  4.144  3.71   2.094
  2.776  2.594  2.754  4.802  4.118  3.71   3.598  3.324  3.254  3.14
  3.868  3.824  2.936  2.206  3.436  3.368  3.368  3.21   1.592  3.186]
Predict X: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1]
Weights: [-0.4  -0.68  1.82]

Process finished with exit code 0
```

Files & Functions

perceptron.py

This file contains code that demonstrates the usage of the Perceptron class.

ML.py

```
CLASSES
  class Perceptron(learning_rate=0.01, iterations=10)
   |
   | init(self, learning_rate=0.01, iterations=10)
   |    Initialize the Perceptron.
   |    Args:
   |       learning_rate Float: used to scale the weight array
   |       iterations Int: number of iterations for fitting data to labels
   |
   | fit(self, X, y)
   |    Fit training data.
   |    Args:
   |       X : Training vectors, X.shape : [#samples, #features]
   |       y : Target values, y.shape : [#samples]
   |
   | net_input(self, X)
   |    Calculate the net input.
   |    Args:
   |       X : Training vectors, X.shape : [#samples, #features]
   |    Returns:
   |       Float: the dot product (X.w) plus the bias
   |
   | predict(self, X)
   |    Return the class label after unit step
   |    Args:
   |       X : Training vectors, X.shape : [#samples, #features]
   |    Returns:
   |       Int: the predicted class label (1 or -1)
   |
   | errors
   |    This is the getter for the error array. Using a getter prevents the caller
   |    from changing the array.
   |    Returns:
   |       list: the array of errors in each iteration
   |
   | weight
   |    This is the getter for the weight array. Using a getter prevents the caller
   |    from changing the array.
   |    Returns:
   |       numpy.ndarray: the current weight array
```

Testing

The Perceptron class was tested using the famous Iris Data Set for machine learning. The expected output of `perceptron.py` is given above and was checked against the output of Dr. Karlsson's program as given in the Perceptron assignment PDF to ensure correctness. Additional tests were done using different pairs and triples in the Iris Data Set. The Perceptron was also tested using all four features in the Iris Data Set to ensure that it is generic enough to accept any number of features, however; it should be noted that when using more than two features from the Iris Data Set the data seems to be inseparable.

Regression Models

Linear Regression Model

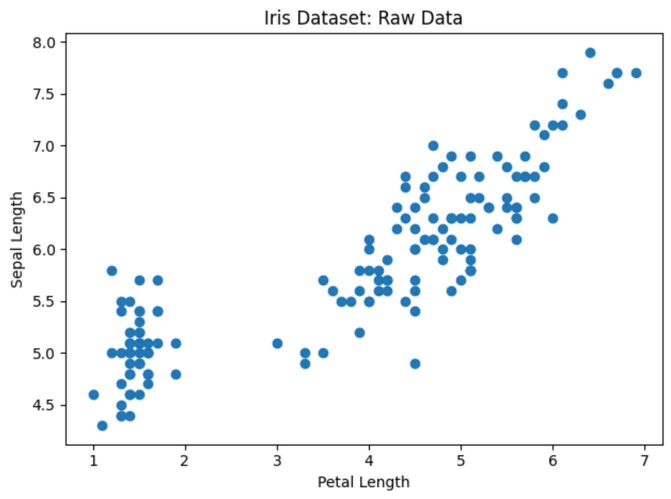
Description

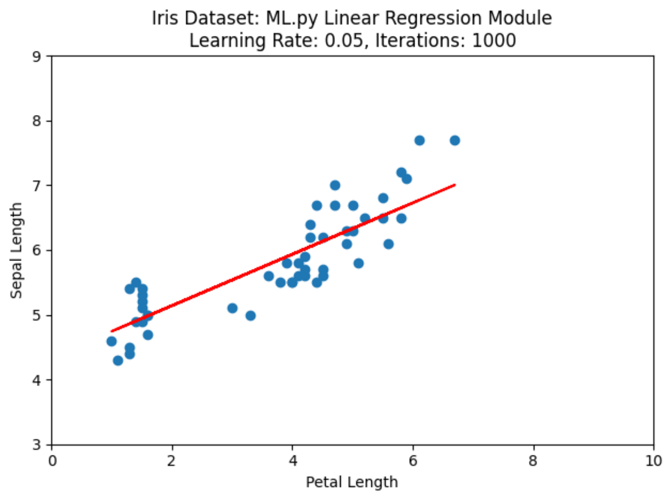
Linear regression is a useful tool for modeling linear relationships between data. The linear regression model included with the `ML.py` library uses stochastic gradient descent to predict a y-value for some given x-value. Since all linear regression problems are convex regardless of the input data, gradient descent can be used without worrying about "getting stuck" in a local minimum. The linear regression model works by iterating over the training data, calculating the predicted y-value, computing the partial derivatives of mean squared error with respect to the slope and y-intercept, and finally updating the weights associated with the slope and y-intercept in order to improve the error on the next iteration. Currently, the model only works for *d=1* dimensional data (i.e. **X** has one *x* value per row).

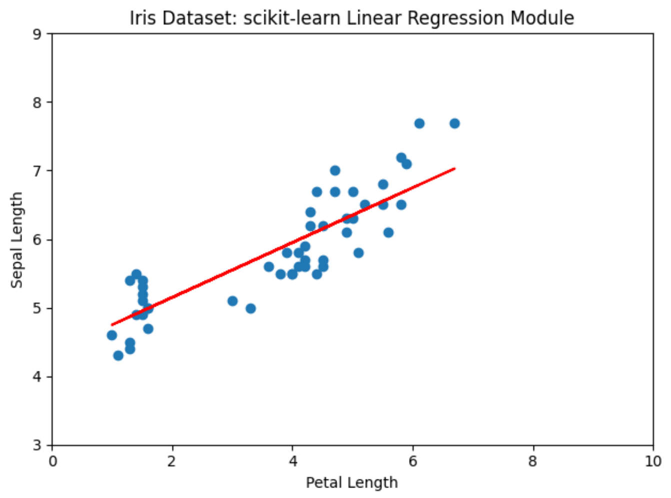
Usage

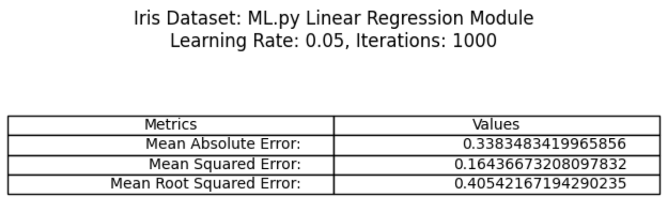
A file called `linearRegression.py` has been included. This file demonstrates the usage of the LinearRegression class from the `ML.py` library. The demonstration uses data from the famous Iris Data Set for machine learning as well as from a fish market dataset. The scikit-learn linear regression model is used to check the accuracy of the `ML.py` linear regression model.

To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 linearRegression.py` in the root directory of the project. Example screenshots of the expected output are provided below.



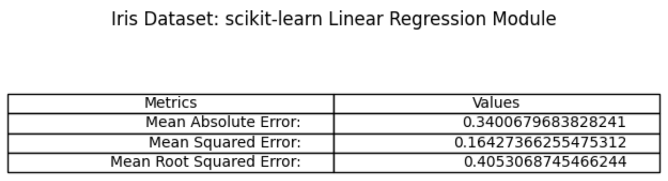






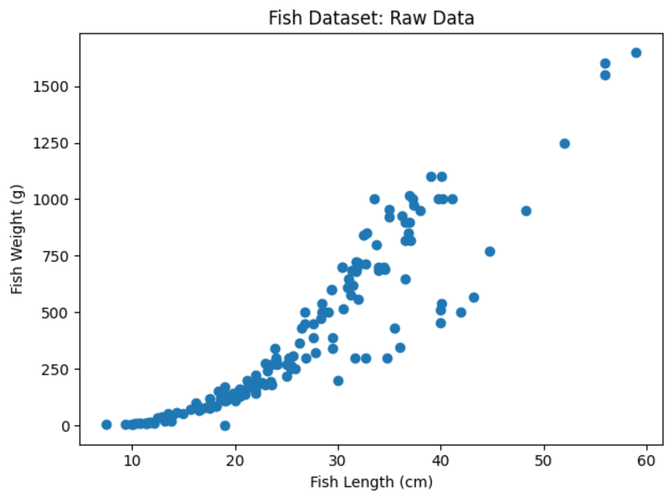
Iris Dataset: ML.py Linear Regression Module
Learning Rate: 0.05, Iterations: 1000

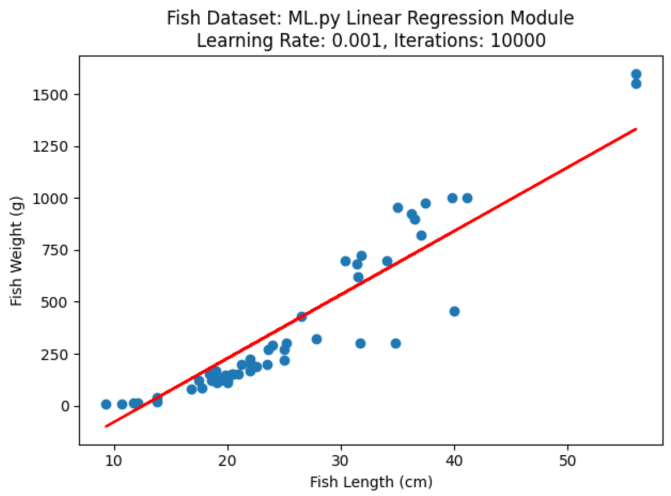
Metrics	Values
Mean Absolute Error:	0.3383483419965856
Mean Squared Error:	0.16436673208097832
Mean Root Squared Error:	0.40542167194290235

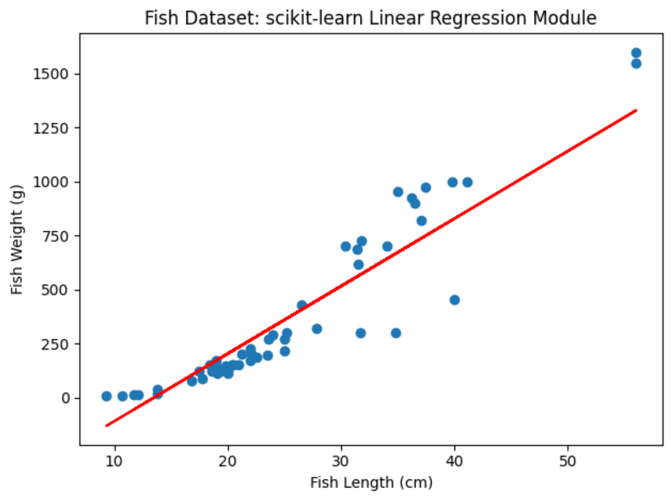


Iris Dataset: scikit-learn Linear Regression Module

Metrics	Values
Mean Absolute Error:	0.3400679683828241
Mean Squared Error:	0.16427366255475312
Mean Root Squared Error:	0.4053068745466244







Fish Dataset: ML.py Linear Regression Module
Learning Rate: 0.001, Iterations: 10000

Metrics	Values
Mean Absolute Error:	112.7518044049906
Mean Squared Error:	19787.248733607346
Mean Root Squared Error:	140.66715584530507

Fish Dataset: scikit-learn Linear Regression Module

Metrics	Values
Mean Absolute Error:	105.33454780144442
Mean Squared Error:	18727.74306453916
Mean Root Squared Error:	136.8493444066838

Files & Functions

linearRegression.py

This file contains code that demonstrates the usage of the LinearRegression class.

ML.py

```
class LinearRegression(builtins.object)
 |  LinearRegression(learning_rate=0.05, iterations=1000)
 |
 |  This is the linear regression implementation for the ML library.
 |
 |  Methods defined here:
 |
 |  __init__(self, learning_rate=0.05, iterations=1000)
 |      Initialize the linear regression module.
 |
 |      Args:
 |          learning_rate (float, optional): used to scale the weight array. Defaults to 0.05.
 |          iterations (int, optional): number of gradient descent iterations. Defaults to 1000.
 |
 |  cost(self, X, Y)
 |      Mean squared error cost function.
 |
 |      Args:
 |          X: X test vector (independent variables)
 |          Y: Y training vector (dependent variables)
 |
 |      Returns:
 |          Mean squared error
 |
 |  fit(self, X, Y)
 |      Fit training data using stochastic gradient descent.
 |
 |      Args:
 |          X : X training vector (independent variables)
 |          Y : Y training vector (dependent variables)
 |
 |  predict(self, X_test)
 |      Return the predicted Y values.
 |
 |      Args:
 |          X_test : X test vector
 |
 |      Returns:
 |          Y_pred : Y prediction vector
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Testing

The LinearRegression class was tested using the famous Iris Data Set for machine learning as well as a fish market dataset. The expected output of `linearRegression.py` is given above. Part of the above output is the output of the scikit-learn linear regression model, which was used to check the correctness of the `ML.py` linear regression model. In addition to performing visual checks of scatter plots and regression lines, mean squared error, mean absolute error, and mean root squared error were compared.

Logistic Regression Model

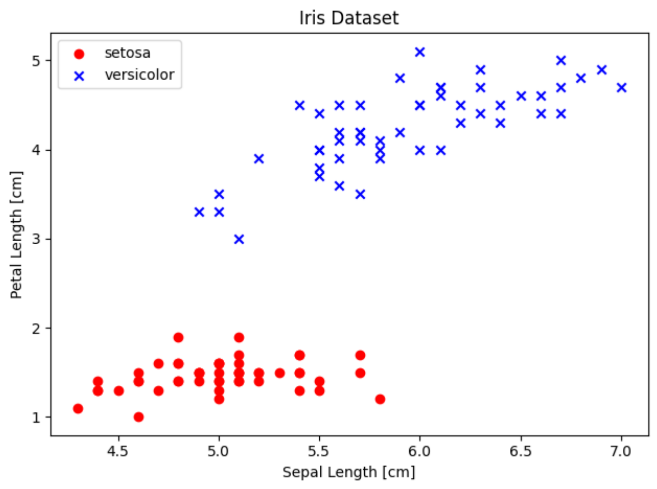
Description

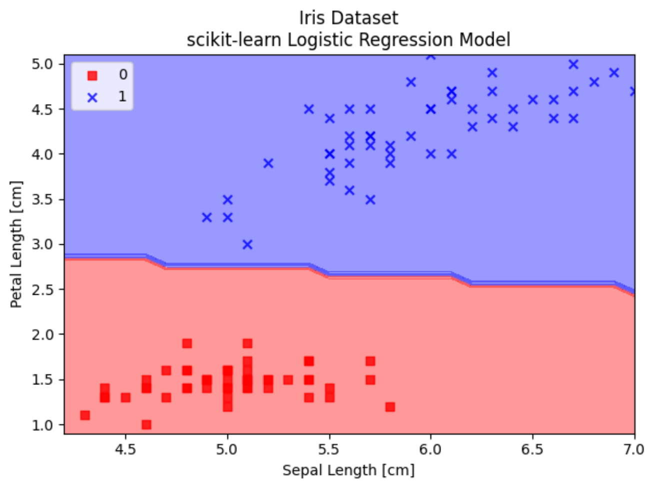
Logistic regression is used to make binary predictions about the likelihood that data belongs to some class. The example below uses logistic regression to predict whether datapoints from the Iris dataset belong to the Iris-setosa class or the Iris-versicolor class. For each datapoint, the likelihood of that datapoint belonging to some class is calculated. This results in a probability, which is then rounded to either 0 or 1. The value of 0 or 1 can then be mapped to a label. Logistic regression works by using a sigmoid function to construct a hypothesis class, which is then used for classifying data. The specific sigmoid function used is the logistic function $f(Z) = \frac{1}{1+\exp(-Z)}$. If X is the input data such that each column of X is associated with some feature, then Z is the result of applying weights to the data associated with each feature in X. The logistic function has the advantage of being convex, so gradient descent can be used to find the global minimum.

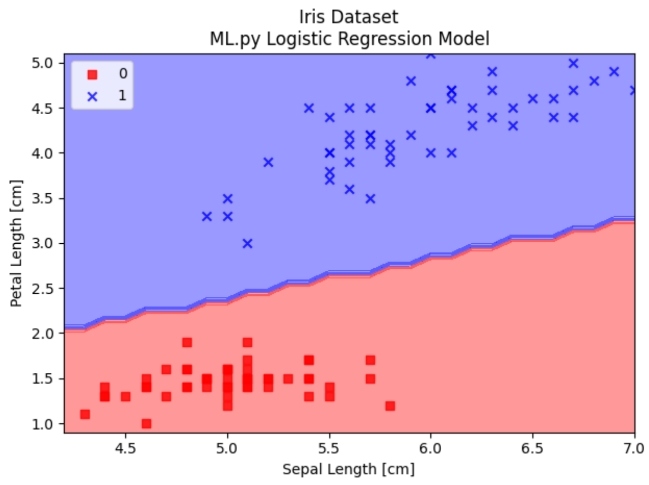
Usage

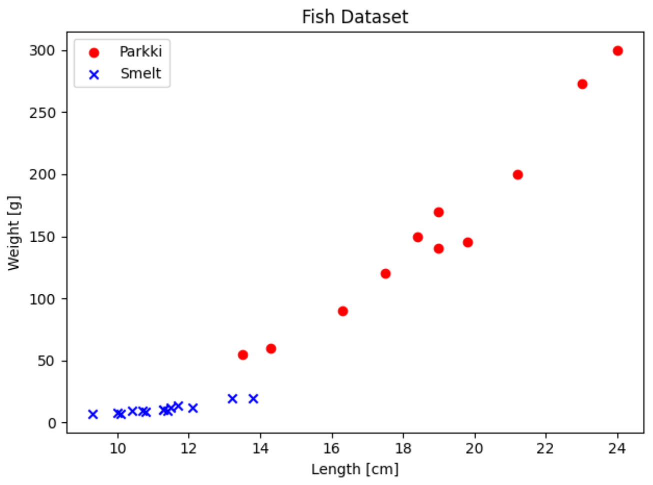
A file called `logisticRegression.py` has been included. This file demonstrates the usage of the LogisticRegression class from the `ML.py` library. The demonstration uses data from the famous Iris Data Set for machine learning as well as from a fish market dataset. The scikit-learn logistic regression model is used to check the accuracy of the `ML.py` logistic regression model.

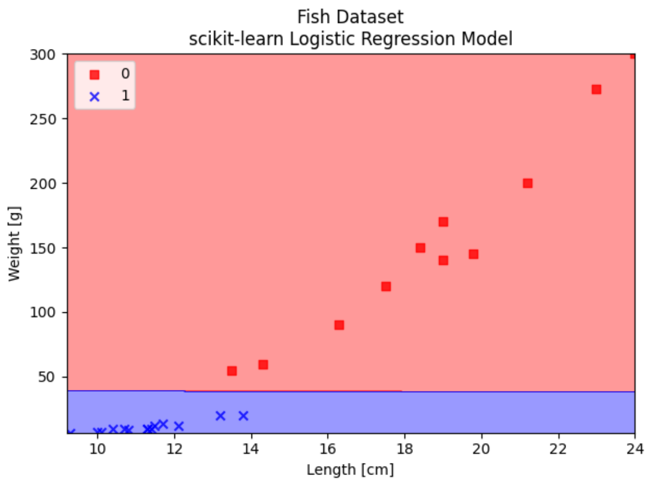
To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 logisticRegression.py` in the root directory of the project. Example screenshots of the expected output are provided below.

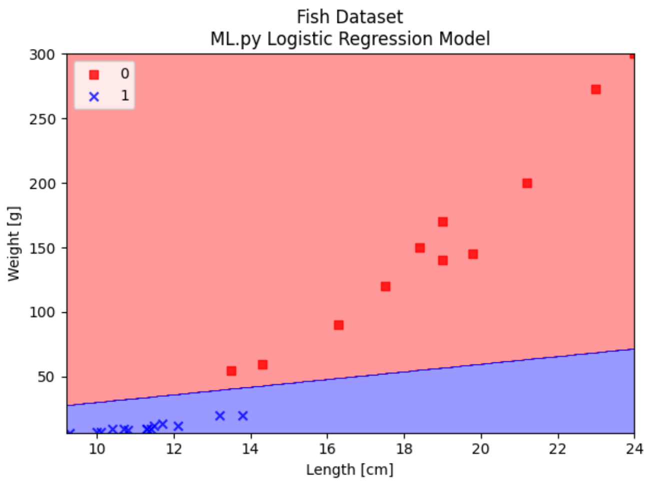












Files & Functions

logisticRegression.py

This file contains code that demonstrates the usage of the LogisticRegression class.

ML.py

```
class LogisticRegression(builtins.object)
 |  LogisticRegression(learning_rate=0.01, iterations=10)
 |
 |  This is the logistic regression implementation for the ML library.
 |
 |  Methods defined here:
 |
 |  __init__(self, learning_rate=0.01, iterations=10)
 |      Initialize the logistic regression module.
 |
 |      Args:
 |          learning_rate (float, optional): used to scale the weight array. Defaults to 0.01.
 |          iterations (int, optional): number of gradient descent iterations. Defaults to 10.
 |
 |  fit(self, X, Y)
 |      Fit training data.
 |
 |      Args:
 |          X: X training vector (independent variables)
 |          Y : Y training vector (dependent variables)
 |
 |  predict(self, X_test)
 |      Return the predicted Y values.
 |
 |      Args:
 |          X_test: X_test : X test vector
 |
 |      Returns:
 |          Y_pred : Y prediction vector
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

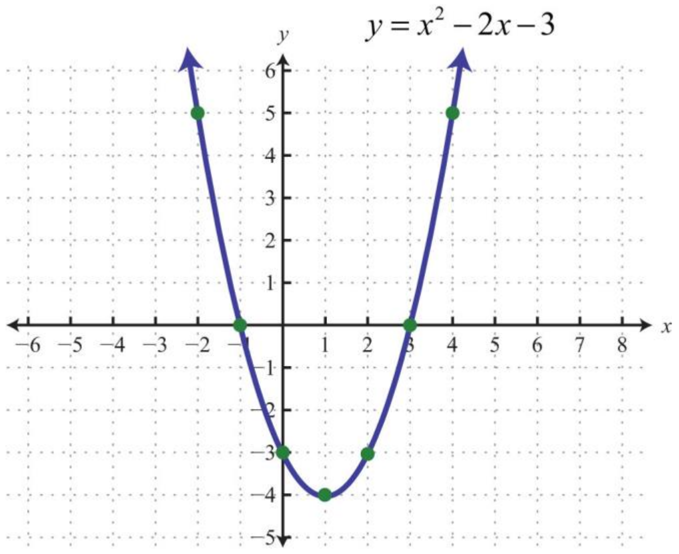
Testing

The LogisticRegression class was tested using the famous Iris Data Set for machine learning as well as a fish market dataset. The expected output of `logisticRegression.py` is given above. Part of the above output is the output of the scikit-learn logistic regression model, which was used to check the correctness of the `ML.py` logistic regression model.

Stochastic Gradient Descent

Description

Gradient descent is an iterative method for minimizing differentiable functions. It works by descending a gradient, or slope, until the lowest point is reached. Consider the function below:



The lowest point of this function is (1, -4) and it is possible to tell when the lowest point of the function is reached by looking at changes to the slope. If we were to start on one end of the function and "walk" down the function towards the middle, we would know we reached a minimum when the slope decreases to zero. What is more difficult to tell is if the minimum reached is the *global* minimum or just a *local* minimum. If the function is convex, then it is guaranteed to be a global minimum. Linear regression is a convex optimization problem, so gradient descent is very often used in linear regression to minimize the error function.

Stochastic gradient descent is a version of gradient descent. Stochastic gradient descent is useful because it converges much faster than normal gradient descent. It achieves this by randomly selecting a sample from the training set during each training iteration and updating the weights accordingly (for 2D linear regression the weights would be slope and intercept). Because samples are randomly selected from the training set, stochastic gradient descent takes a more direct, albeit noisy, path to the global minimum. Gradient descent, on the other hand, uses all of the available training data during each iteration to update the weights, so it is slower to converge, but its path to the optimum is less noisy.

Usage

Stochastic gradient descent is used by the linear regression model included with the `ML.py` library. For more information see Linear Regression Model.

Hypothesis Classes of Low VC Dimension

Axis-Aligned Rectangles

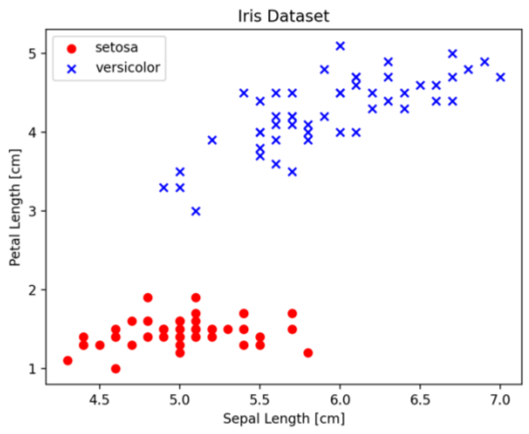
Description

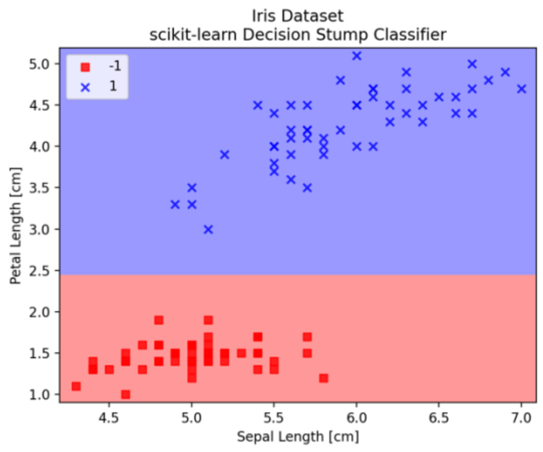
Axis-aligned rectangles are a type of low VC dimension hypothesis class that work by using training data to form a bounding box separating positively labelled points from negatively labelled points. Test data is then classified based on whether or not the data point is within the axis-aligned rectangle. Data points within the axis-aligned rectangle are labelled positively and those outside of the axis-aligned rectangle are labelled negatively. If the input data is linearly separable, the dimensions of the axis-aligned rectangle are very simple to compute. For each column (feature) in **X**, the min and max of that column are points on the perimeter of the bounding box. If the data is not linearly separable, the axis-aligned rectangle will be resized to minimize the total number of misclassifications.

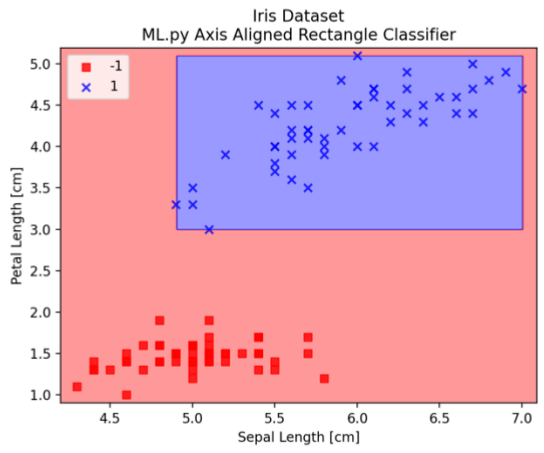
Usage

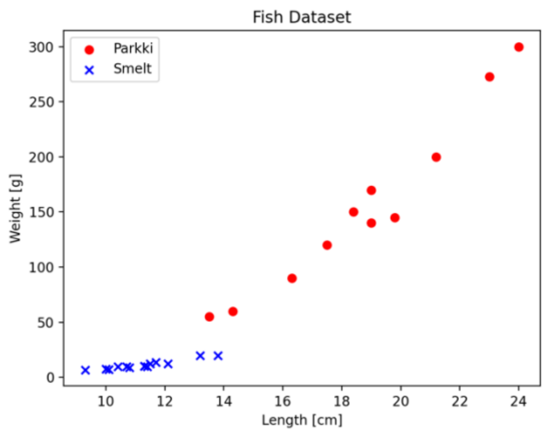
A file called `axisAlignedRectangle.py` has been included. This file demonstrates the usage of the AxisAlignedRectangles class from the `ML.py` library. The demonstration uses data from the famous [Iris Data Set] for machine learning as well as from a [fish market dataset]. The [scikit-learn AdaBoostClassifier] is used to boost the performance of the low VC dimension classifier and enable easy comparisons with other types of classifiers.

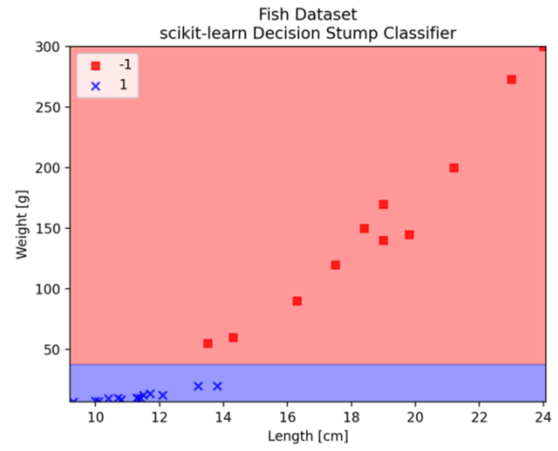
To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 axisAlignedRectangle.py` in the root directory of the project. Example screenshots of the expected output are provided below.

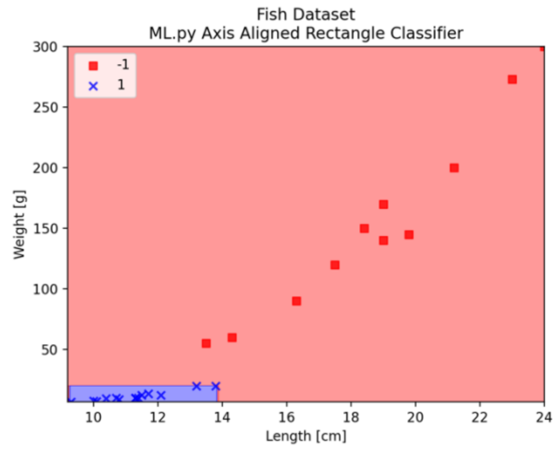


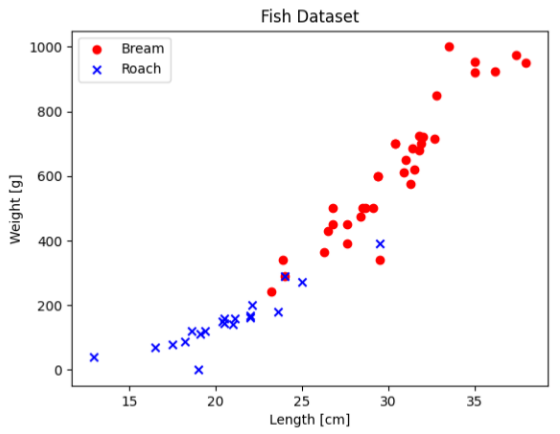


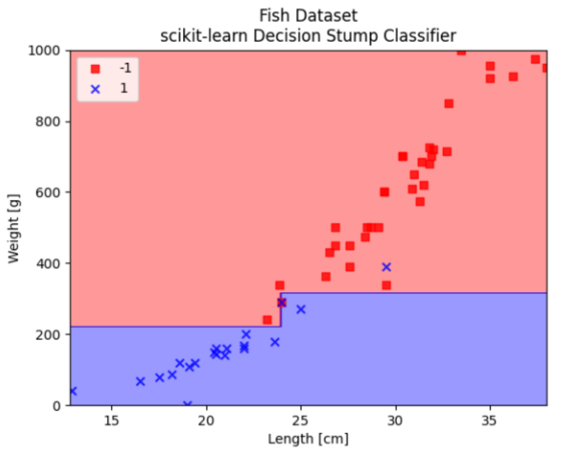


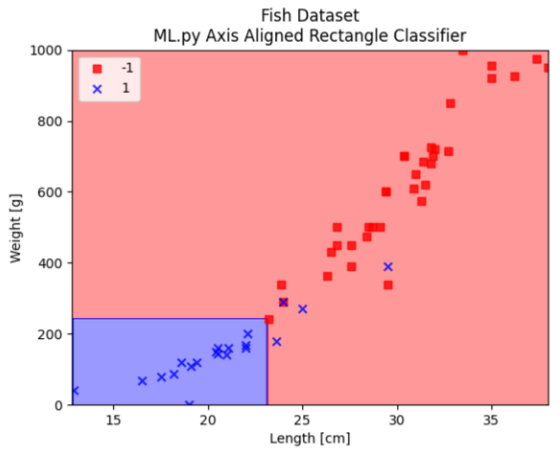


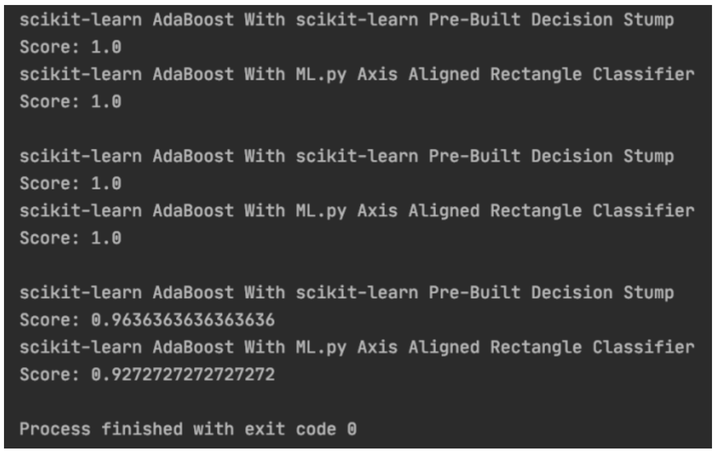












```
scikit-learn AdaBoost With scikit-learn Pre-Built Decision Stump
Score: 1.0
scikit-learn AdaBoost With ML.py Axis Aligned Rectangle Classifier
Score: 1.0

scikit-learn AdaBoost With scikit-learn Pre-Built Decision Stump
Score: 1.0
scikit-learn AdaBoost With ML.py Axis Aligned Rectangle Classifier
Score: 1.0

scikit-learn AdaBoost With scikit-learn Pre-Built Decision Stump
Score: 0.9636363636363636
scikit-learn AdaBoost With ML.py Axis Aligned Rectangle Classifier
Score: 0.9272727272727272

Process finished with exit code 0
```

Files & Functions

axisAlignedRectangle.py

This file contains code that demonstrates the usage of the AxisAlignedRectangles class.

ML.py

```
class AxisAlignedRectangles(sklearn.base.BaseEstimator, sklearn.base.ClassifierMixin)
 |  AxisAlignedRectangles(iterations=10)
 |
 |  This is the axis-aligned rectangle low vc dimension learner implementation
 |     for the ML library.
 |
 |  Args:
 |      BaseEstimator : Base class for all estimators in scikit-learn,
 |                      used for compatibility with the sci-kit-learn AdaBoostClassifier
 |     ClassifierMixin : Mixin class for all classifiers in scikit-learn,
 |                     used for compatibility with the sci-kit-learn AdaBoostClassifier
 |
 |  Method resolution order:
 |     AxisAlignedRectangles
 |     sklearn.base.BaseEstimator
 |     sklearn.base.ClassifierMixin
 |     builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, iterations=10)
 |     Initialize the axis-aligned rectangle low vc dimension learner.
 |
 |     Args:
 |        iterations: number of iterations for reducing size of axis-aligned rectangle, defaults to 10
 |
 |  fit(self, X, y, sample_weight=None)
 |     Fit training data.
 |
 |     Args:
 |        X : X training vector
 |        y : y label vector
 |        sample_weight (optional): Required for compatibility with the scikit-learn Adaboost module. Defaults to None.
 |
 |     Returns:
 |        self : Required for compatibility with the scikit-learn Adaboost module.
 |
 |  predict(self, X)
 |     Return the predicted Y values.
 |
 |     Args:
 |        X : X test vector
 |
 |     Returns:
 |        Y_pred : Y prediction vector
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |     Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |     Get parameters for this estimator.
 |
 |     Parameters
 |     ----------
 |     deep : bool, default=True
 |        If True, will return the parameters for this estimator and
 |        contained subobjects that are estimators.
 |
 |     Returns
 |     -------
```

```
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.BaseEstimator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample that
 |      each label set be correctly predicted.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for X.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of self.predict(X) wrt. y.
```

Testing

The AxisAlignedRectangles class was tested using the famous Iris Data Set for machine learning as well as a fish market dataset. The expected output of `axisAlignedRectangle.py` is given above. The scikit-learn AdaBoostClassifier was used to enable easy testing because it allows a base estimator (i.e. low VC dimension learner) to be passed in as a parameter. This enabled comparisons to be made between the axis-aligned rectangle classifier and the built-in scikit-learn decision stump classifier. The results of the comparison are reflected in the output shown above.

Decision Stump

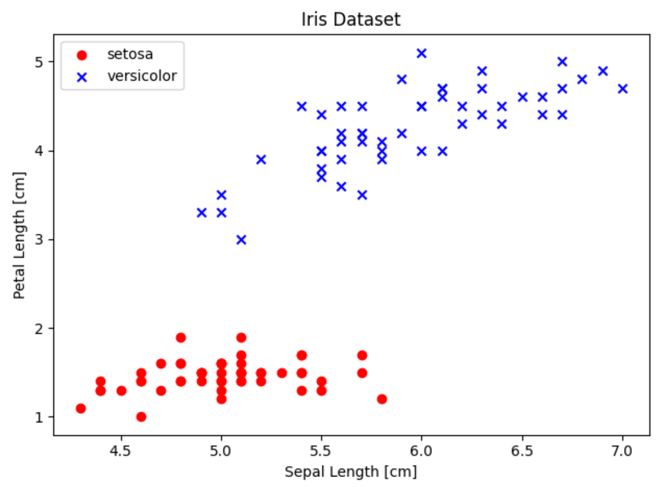
Description

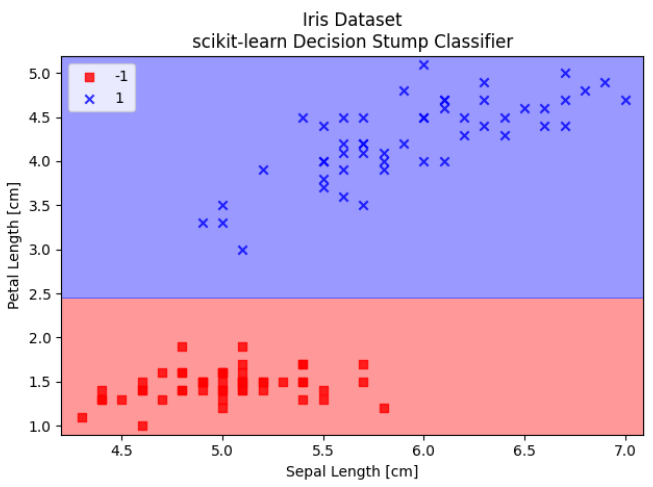
Decision stumps are a type of low VC dimension hypothesis class and can be thought of as a one-level decision tree. A decision stump works by selecting a single feature from a set of training data and attaching a threshold to that feature. The decision stump then uses that feature-threshold pair to label the test data. The `ML.py` decision stump selects the feature-threshold pair that yields the smallest error over the training data. Part of this selection process includes attaching an inequality lambda to the feature-threshold pair. The inequality lambda can be greater-than or less-than and is essential for predicting labels. The reasoning for this is that the feature-threshold pair alone do not provide enough information to make a comparison. We must also know whether the value of the feature should be above or below the threshold.

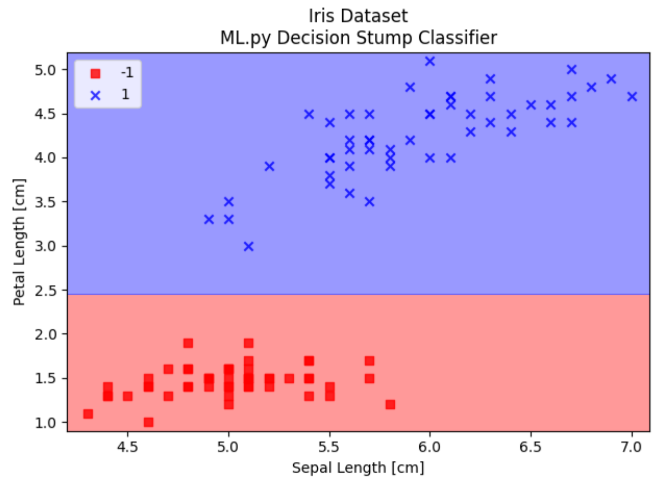
Usage

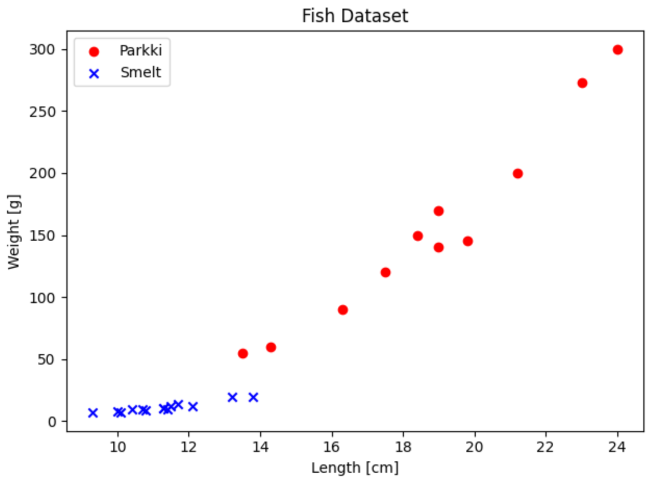
A file called `decisionStump.py` has been included. This file demonstrates the usage of the DecisionStump class from the `ML.py` library. The demonstration uses data from the famous Iris Data Set for machine learning as well as from a fish market dataset. The scikit-learn AdaBoostClassifier is used to boost the performance of the low VC dimension classifier and enable easy comparisons with other types of classifiers.

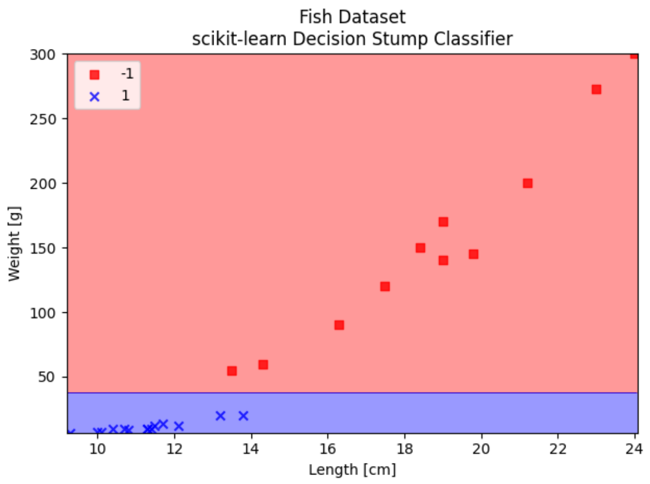
To run the sample code first make sure that all of the dependencies are installed. Next, open a terminal and run `python3 decisionStump.py` in the root directory of the project. Example screenshots of the expected output are provided below.

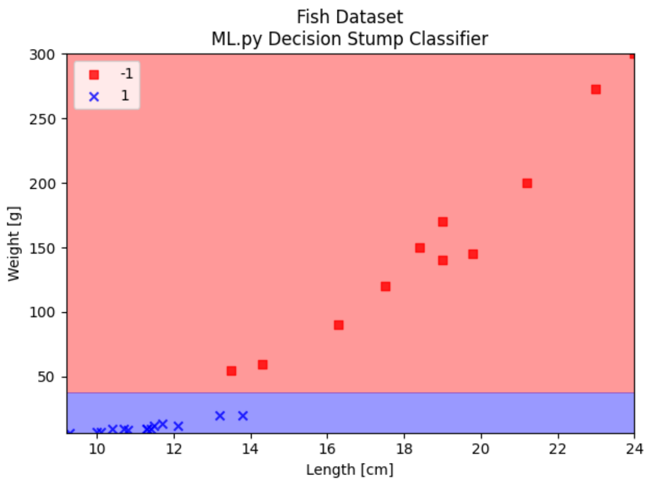


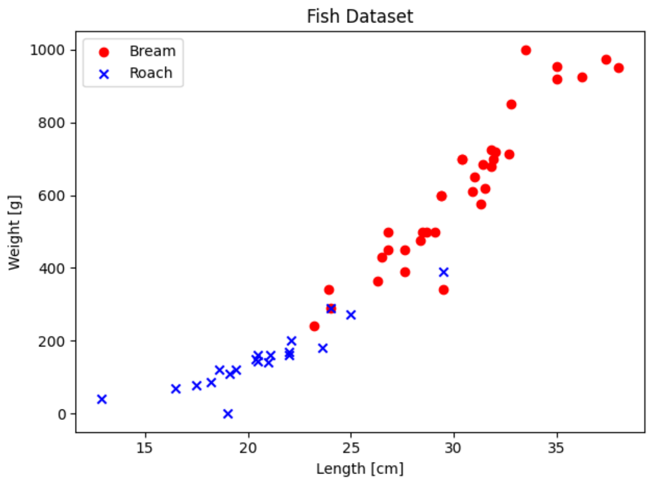


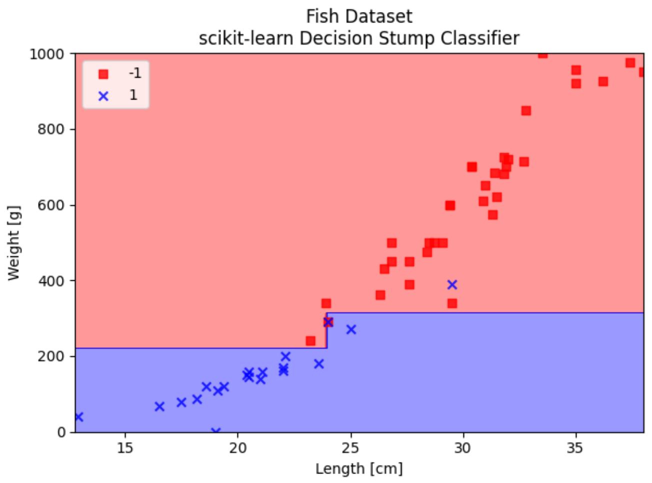


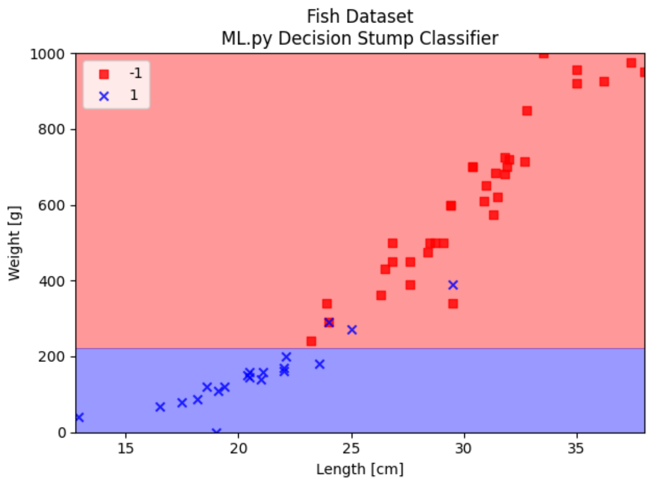


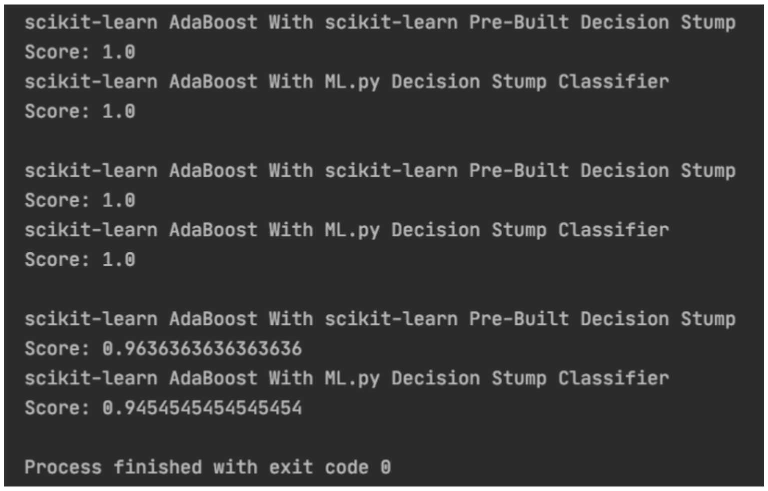











```
scikit-learn AdaBoost With scikit-learn Pre-Built Decision Stump
Score: 1.0
scikit-learn AdaBoost With ML.py Decision Stump Classifier
Score: 1.0

scikit-learn AdaBoost With scikit-learn Pre-Built Decision Stump
Score: 1.0
scikit-learn AdaBoost With ML.py Decision Stump Classifier
Score: 1.0

scikit-learn AdaBoost With scikit-learn Pre-Built Decision Stump
Score: 0.9636363636363636
scikit-learn AdaBoost With ML.py Decision Stump Classifier
Score: 0.9454545454545454

Process finished with exit code 0
```

Files & Functions

decisionStump.py

This file contains code that demonstrates the usage of the DecisionStump class.

ML.py

```
class DecisionStump(sklearn.base.BaseEstimator, sklearn.base.ClassifierMixin)
 |  This is the decision stump low vc dimension learner implementation
 |    for the ML library.
 |
 |  Args:
 |     BaseEstimator : Base class for all estimators in scikit-learn,
 |                 used for compatibility with the sci-kit-learn AdaBoostClassifier
 |     ClassifierMixin : Mixin class for all classifiers in scikit-learn,
 |                  used for compatibility with the sci-kit-learn AdaBoostClassifier
 |
 |  Method resolution order:
 |     DecisionStump
 |     sklearn.base.BaseEstimator
 |     sklearn.base.ClassifierMixin
 |     builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |     Initialize the decision stump low vc dimension learner.
 |
 |  fit(self, X, y, sample_weight=None)
 |     Fit training data.
 |
 |     Args:
 |        X : X training vector
 |        y : y label vector
 |        sample_weight (optional): Required for compatibility with the scikit-learn Adaboost module. Defaults to None.
 |
 |     Returns:
 |        self : Required for compatibility with the scikit-learn Adaboost module.
 |
 |  predict(self, X)
 |     Return the predicted Y values.
 |
 |     Args:
 |        X : X test vector
 |
 |     Returns:
 |        Y_pred : Y prediction vector
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |     Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |     Get parameters for this estimator.
 |
 |     Parameters
 |     ----------
 |     deep : bool, default=True
 |        If True, will return the parameters for this estimator and
 |        contained subobjects that are estimators.
 |
 |     Returns
 |     -------
 |     params : mapping of string to any
 |        Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.BaseEstimator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample that
 |      each label set be correctly predicted.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for X.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of self.predict(X) wrt. y.
```

Testing

The DecisionStump class was tested using the famous Iris Data Set for machine learning as well as a fish market dataset. The expected output of `decisionStump.py` is given above. The scikit-learn AdaBoostClassifier was used to enable easy testing because it allows a base estimator (i.e. low VC dimension learner) to be passed in as a parameter. This enabled comparisons to be made between the `ML.py` decision stump classifier and the built-in scikit-learn decision stump classifier. The results of the comparison are reflected in the output shown above.

Visualization Modules

Decision Boundary

Description

The `plot_decision_regions` function is used to display the decision regions of a classifier. The decision regions of the classifier are the partitions between different classes of data. The plot below shows the output of this function.

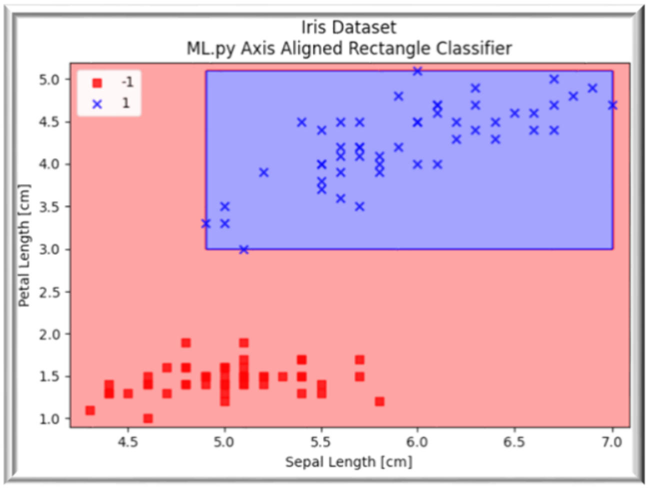


Figure 1: Setosa (-1) is represented by red squares. Versicolor (1) is represented by blue crosses.

Usage

The `plot_decision_regions` function can be imported from the `ML.py` library using the following import statement:

```
from ML import plot_decision_regions
```

Several of the included demo scripts use the `plot_decision_regions` function including `perceptron.py` and `axisAlignedRectangle.py`.

Files & Functions

ML.py

```
FUNCTIONS
  plot_decision_regions(X, y, classifier, resolution=0.02, x_label='', y_label='', title='')
    This is a helper function to plot the decision regions of the classifier. This shows the partition(s) between the
    different classes of objects.

    Args:
      X : Training vectors, X.shape : [#samples, #features]
      y : Target values, y.shape : [#samples]
      classifier : the classification algorithm
      resolution (float, optional) : the resolution of the meshgrid
      x_label (string, optional) : the x label for the plot
      y_label (string, optional) : the y label for the plot
      title (string, optional) : the title for the plot
```

Regression Line

Description

Regression lines help to visualize the relationship between x, y points. Regression lines are typically added to a scatter plot so that the relationship between the points and the line is clear. The line is a result of linear regression and the equation of the regression line is used to approximate the corresponding y-value for a given x-value.

Usage

The `plot_regression_line` function can be imported from the `ML.py` library using the following import statement:

```
from ML import plot_regression_line
```

The `linearRegression.py` demo script uses this function to display the results of linear regression, as shown below.

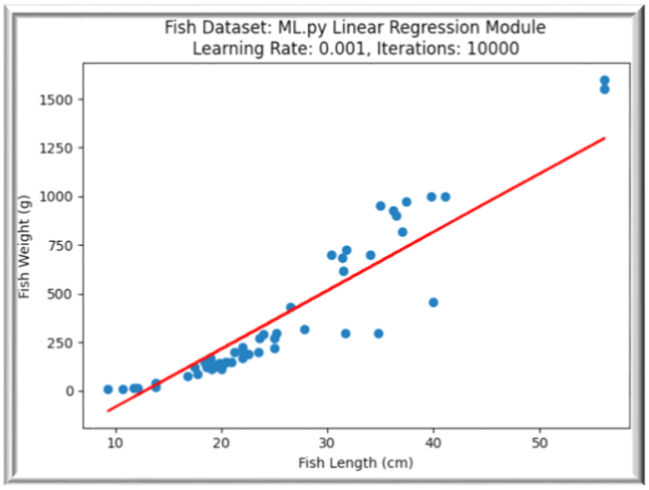


Figure 2: Sample Regression Line

Files & Functions

ML.py

FUNCTIONS

plot_regression_line(y_predicted, x_actual, y_actual, x_label='', y_label='', title='', line_color='red', x_range=None, y_range=None)

This function plots the linear regression line.

A linear regression line has an equation of the form Y = a + bX.
X is the explanatory variable and Y is the dependent variable.
The slope of the line is b, and a is the intercept.

Args:

y_predicted : the y values predicted by the linear regression learner
x_actual : the actual x values
y_actual : the actual y values
x_label (str, optional): Horizontal axis label. Defaults to "".
y_label (str, optional): Vertical axis label. Defaults to "".
title (str, optional): Plot title. Defaults to "".
line_color (str, optional): Plot line color. Defaults to 'red'.
x_range (tuple, optional): x range of graph
y_range (tuple, optional): y range of graph

Scatter Plots

Description

Scatter plots are helpful tools to observe the relationship between different variables. Scatter plots can also be used to help make determinations about whether data is separable or inseparable.

Usage

Matplotlib contains a function called `matplotlib.pyplot.scatter` that will display a scatter plot. Usage of this function is described in the Matplotlib documentation and demonstrated in each of the demo scripts included with the project.

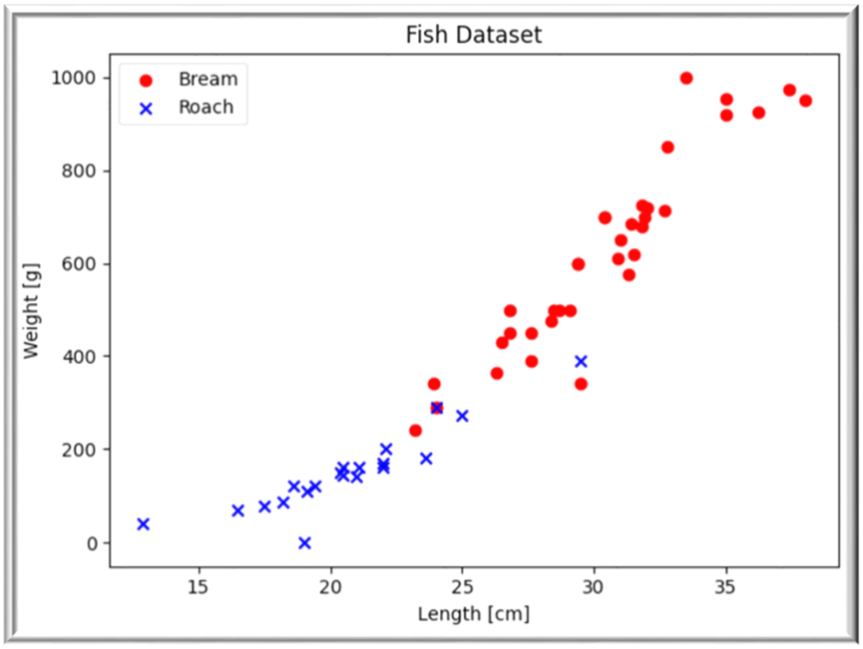


Figure 3: Sample scatter plot using data from the Fish Market dataset.

References

Algorithms

Decision Stumps
Gradient Descent
Hinge Loss
k-Nearest Neighbors
Logistic Regression
Perceptron
Stochastic Gradient Descent

Data Sets

Iris Data Set
Fish Market Data Set

ML Resources

matplotlib
Scatter Plots

Python
Virtual Environments
Support Vector Machine Plot Function by Jake VanderPlas

scikit-learn
AdaBoost Classifier
Base Estimator
k-Neighbors Classifier
Linear Regression Model
Logistic Regression Model
Support Vector Classifier