BINARY CLASSIFICATION (PERCEPTRON LEARNING AND LOGISTIC REGRESSION)

Ron Michael V. Acda
Activity 8

Objectives

01

Describe perceptron learning and logistic regression

02

Perform binary classification on datasets

Perceptron Algorithm

- The perceptron classification algorithm assumes that the data points are linearly separable; that is, there is a line/plane/hyperplane separating the clusters of data. That is, we can draw a hyperplane of the form:

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n = 0$$

Where n is the number of features and w_i is the weight of the ith feature.

- The equation describes the decision boundary which separates the datapoint clusters. The weights are chosen randomly at first and continually refined by iterating through the samples.

```
class Perceptron:

    def __init__(self, max_iters=100, eta=0.5):
        self.max_iters = max_iters
        self.eta = eta

    def set_trainData(self, X_train, y_train):
        self.X_ = X_train
        d = np.copy(y_train)
        d[d<=0] = 0
        self.y_ = d
        return self

    def fit(self):
        self.w_ = np.zeros(1 + self.X_.shape[1])
        self.errors_ = []

        for _ in range(self.max_iters):
            error = 0
            delta_wj = np.copy(self.w_)

            # Shuffle the training data
            indices = np.arange(len(self.X_))
            np.random.shuffle(indices)
            X_shuffled = self.X_[indices]
            y_shuffled = self.y_[indices]

            for xj, di in zip(X_shuffled, y_shuffled):
                update = self.eta * (di - self.predict(xj))
                self.w_[0] += update
                self.w_[1:] += update * xj
                error += np.abs(update / self.eta)
            self.errors_.append(error)

        return self

    def weights(self):
        return self.w_

    def errors(self):
        return self.errors_

    def predict(self, X):
        a=np.dot(X, self.w_[1:]) + self.w_[0]
        return np.where(a >= 0, 1, 0)
```

Perceptron Algorithm

Here's the algorithm for the perceptron classifier.

1. Initialize an arbitrary weight vector w_ = np.zeros(1 + X_.shape[1]). Note that if there are n features, the weight vector would be a (n+1) x 1 column vector, with the zeroth row as the bias.
2. Iterate throughout the data points (optional: shuffle datapoints first).
    1. For the ith data point, let di and xj be the data label (1 or -1) and feature vector with bias.
    2. Let eta be the learning rate.
    3. Obtain the activation value a = np.dot(X, w_[1:]) + w_[0]
    4. Use the step function as thresholding, b = np.where(a>= 0, 1, 0); that is, return 1 if a>= 0, 0 otherwise.
    5. Let the update u be eta*(di – b) for each data point
    6. For the zeroth row of the weight vector, w_[0] += u
    7. For the other rows, w_[1:] += update* xj
3. A pass through all data points is defined as a single epoch. Repeat Step 2 until the max number of epochs is attained.

Perceptron Algorithm

Note that the higher the learning rate eta, the faster the weights change. This can mean that the algorithm can converge to a local minimum quickly, but also runs the risk of overshooting and not finding a good solution.

On the other hand, a lower learning rate means less risk of overshooting, but the number of iterations or epochs needed to converge to a local minimum is higher.

Perceptron Algorithm

Sometimes, the algorithm may get stuck in local minima that are quite far from the optimal solution (think of valleys or troughs). To avoid this, **we can shuffle the training data every iteration**, introducing a "random" factor that can sometimes "kick" a stuck solution out of an unwanted local minimum.

Disadvantage: Somewhat slows down the program.

Logistic Regression

We can modify the perceptron learning algorithm by returning a continuous value between 0 and 1 for the output.

Instead of passing a through a step function, we can pass it through a sigmoid activation function. This basically “squishes” the real number line into the open interval (0,1).

Therefore, the output is probabilistic i.e., the probability that a point has a certain class label.

Logistic Regression

```
class LogisticRegression:

    def __init__(self, max_iters=100, eta=0.5):
        self.max_iters = max_iters
        self.eta = eta
    def set_trainData(self, X_train, y_train):
        self.X_ = X_train
        d = np.copy(y_train)
        d[d<=0] = 0
        self.y_ = d
        return self
    def fit(self, beta, threshold=None):
        self.w_ = np.zeros(1 + self.X_.shape[1])
        for _ in range(self.max_iters):
            delta_wj = np.copy(self.w_)
            # Shuffle the training data
            indices = np.arange(len(self.X_))
            np.random.shuffle(indices)
            X_shuffled = self.X_[indices]
            y_shuffled = self.y_[indices]
            for xj, di in zip(X_shuffled, y_shuffled):
                update = self.eta * (di - self.predict(xj, beta=beta, threshold=threshold))
                self.w_[0] += update
                self.w_[1:] += update * xj
        return self
    def weights(self):
        return self.w_
    def predict(self, X, beta, threshold=None):
        a = np.dot(X, self.w_[1:]) + self.w_[0]
        z = 1 / (1 + np.exp(-beta * a))
        if threshold is not None:
            return np.where(z >= threshold, 1, 0)
        else:
            return z
```

```
class Perceptron:

    def __init__(self, max_iters=100, eta=0.5):
        self.max_iters = max_iters
        self.eta = eta
    def set_trainData(self, X_train, y_train):
        self.X_ = X_train
        d = np.copy(y_train)
        d[d<=0] = 0
        self.y_ = d
        return self
    def fit(self):
        self.w_ = np.zeros(1 + self.X_.shape[1])
        for _ in range(self.max_iters):
            delta_wj = np.copy(self.w_)
            # Shuffle the training data
            indices = np.arange(len(self.X_))
            np.random.shuffle(indices)
            X_shuffled = self.X_[indices]
            y_shuffled = self.y_[indices]
            for xj, di in zip(X_shuffled, y_shuffled):
                update = self.eta * (di - self.predict(xj))
                self.w_[0] += update
                self.w_[1:] += update * xj
        return self
    def weights(self):
        return self.w_
    def predict(self, X):
        a=np.dot(X, self.w_[1:]) + self.w_[0]
        return np.where(a >= 0, 1, 0)
```

Note that they only differ in the activation function

Dataset 1: Synthetic Dataset

As a first test, we load a synthetic binary classification dataset using sklearn

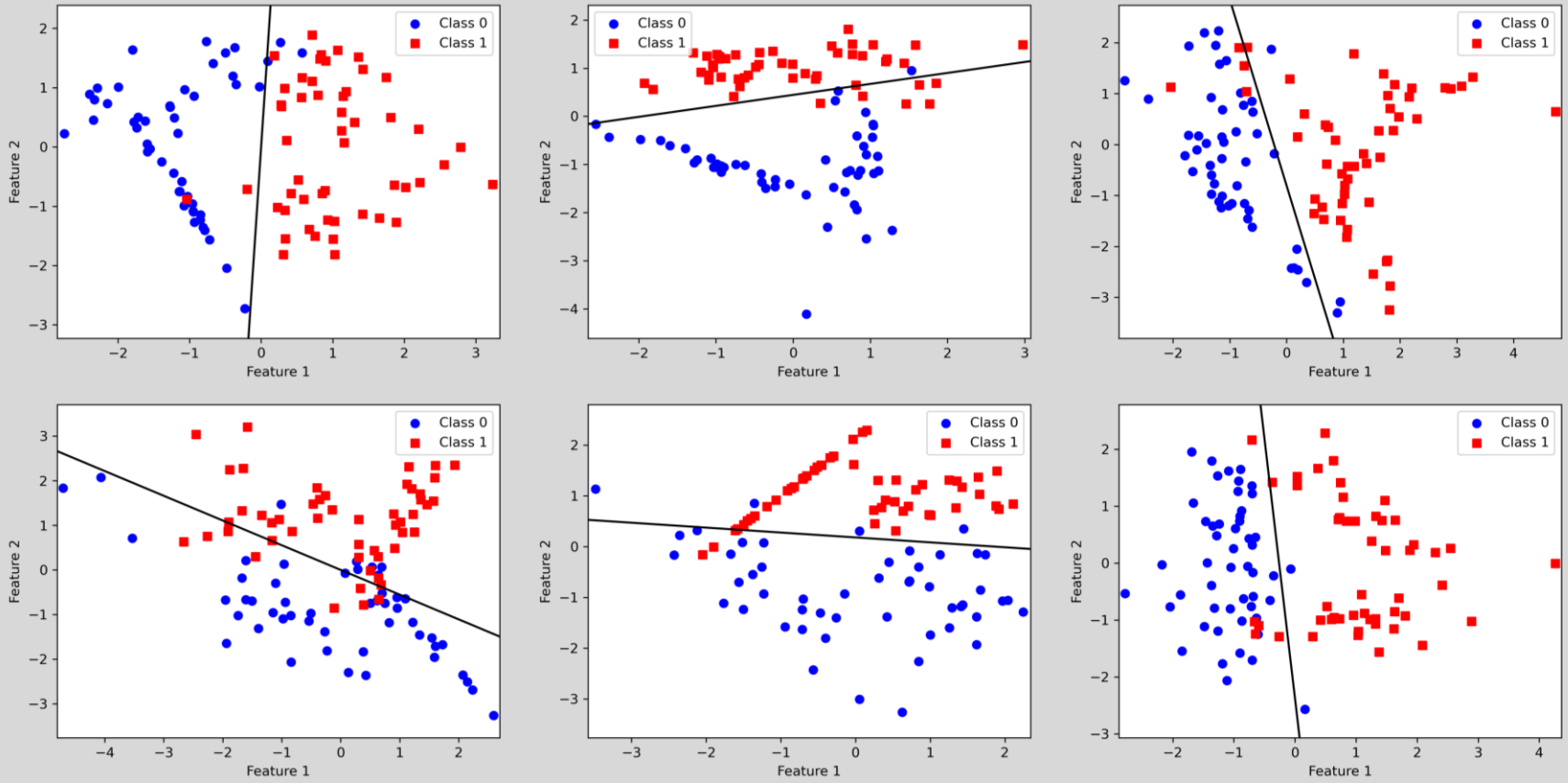
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, random_state=10)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Here, 100 points are loaded, with 2 features. We split the dataset into training and test data (default: 75% training data, 25% test data)

Dataset 1: Synthetic Dataset (Perceptron)

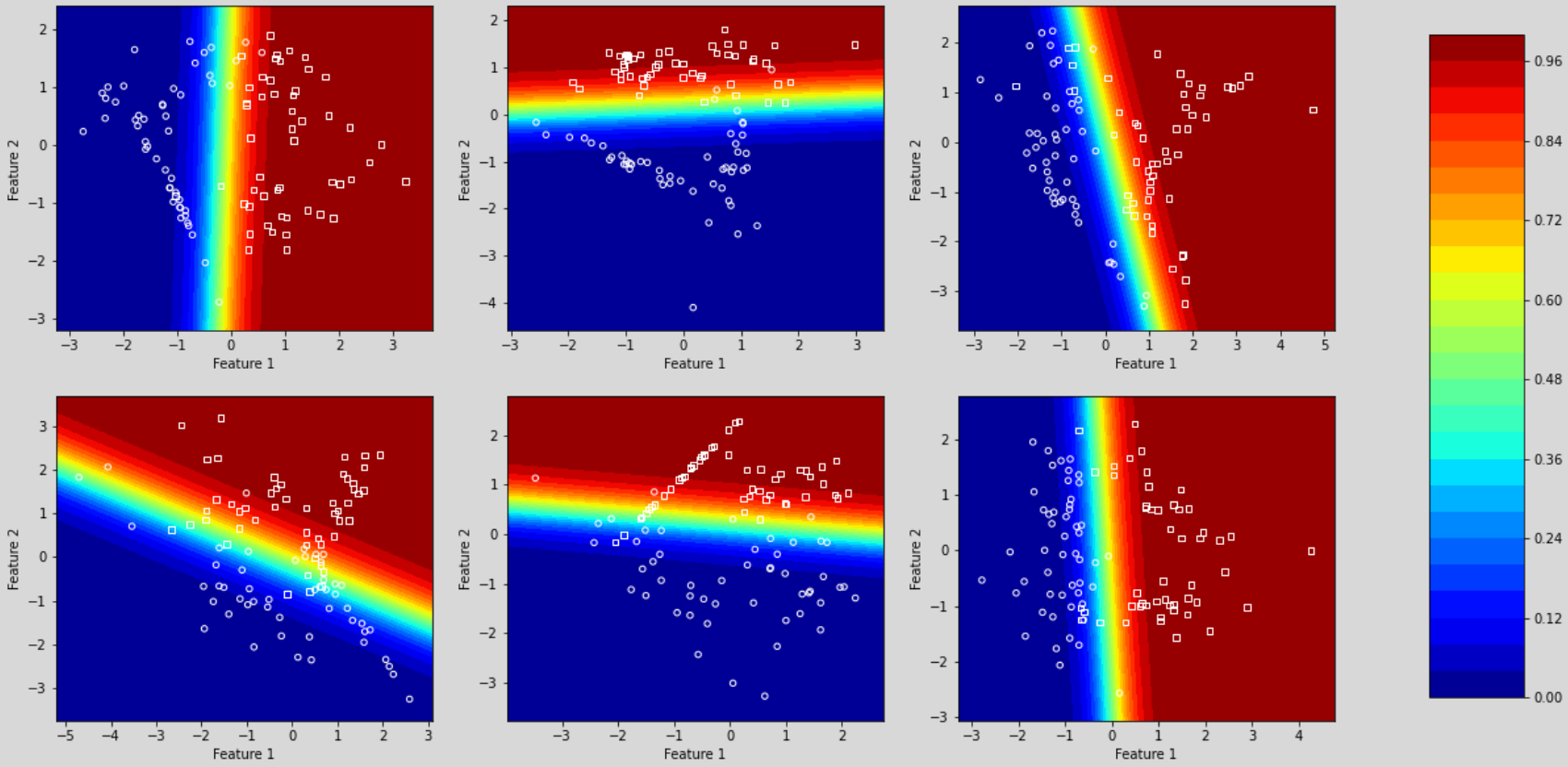
eta=0.05, epochs = 100



The data are linearly separable, so the perceptron decision boundary gives good separation between the two classes, as expected.

Dataset 1: Synthetic Dataset (Logistic Regression)

eta=0.05, epochs = 100



Note that the green line is the perceptron decision boundary. The farther away the point is from the decision boundary, the higher the probability that it belongs to that certain class. The boundary can be thought of as a region of greatest uncertainty in classification.

Dataset 2: Breast Cancer Dataset

Now to test our classifier in a real-life dataset from UCI ML Breast Cancer Wisconsin, classifying tumors as either malignant or benign.

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("Shape of cancer data: {}".format(cancer.data.shape))
print("Sample counts per class:\n{}".format( {n: v for n, v in zip(cancer.target_names,
np.bincount(cancer.target))}))
```

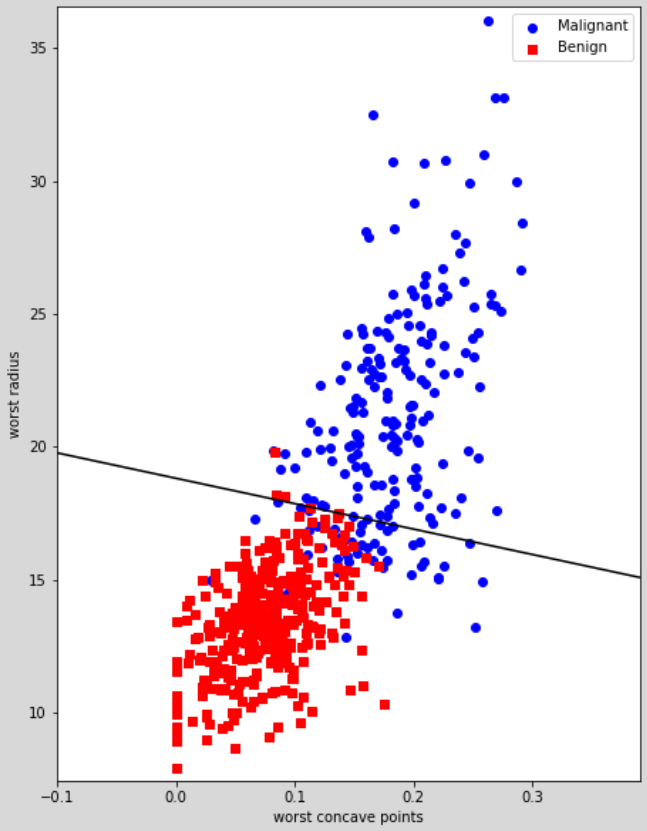
Output:

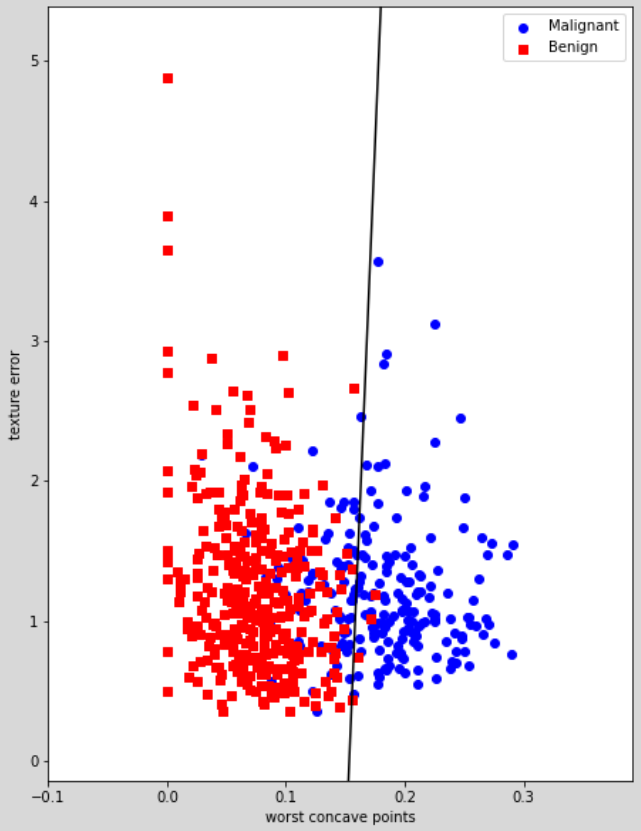
```
Shape of cancer data: (569, 30)
Sample counts per class: {'malignant': 212, 'benign': 357}
```

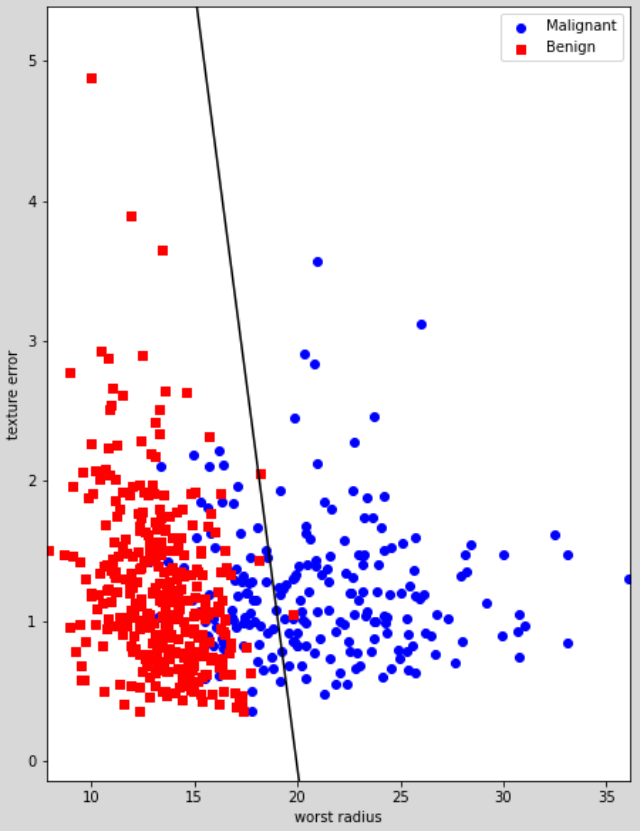
Dataset 2: Breast Cancer Dataset (Perceptron)

There are 30 features in the dataset, but I only chose (1) worst radius, (2) worst concave points and (3) texture error. Note that by selecting these features, we can see that there is good linear separation between data points.

Eta = 0.01, epochs = 300



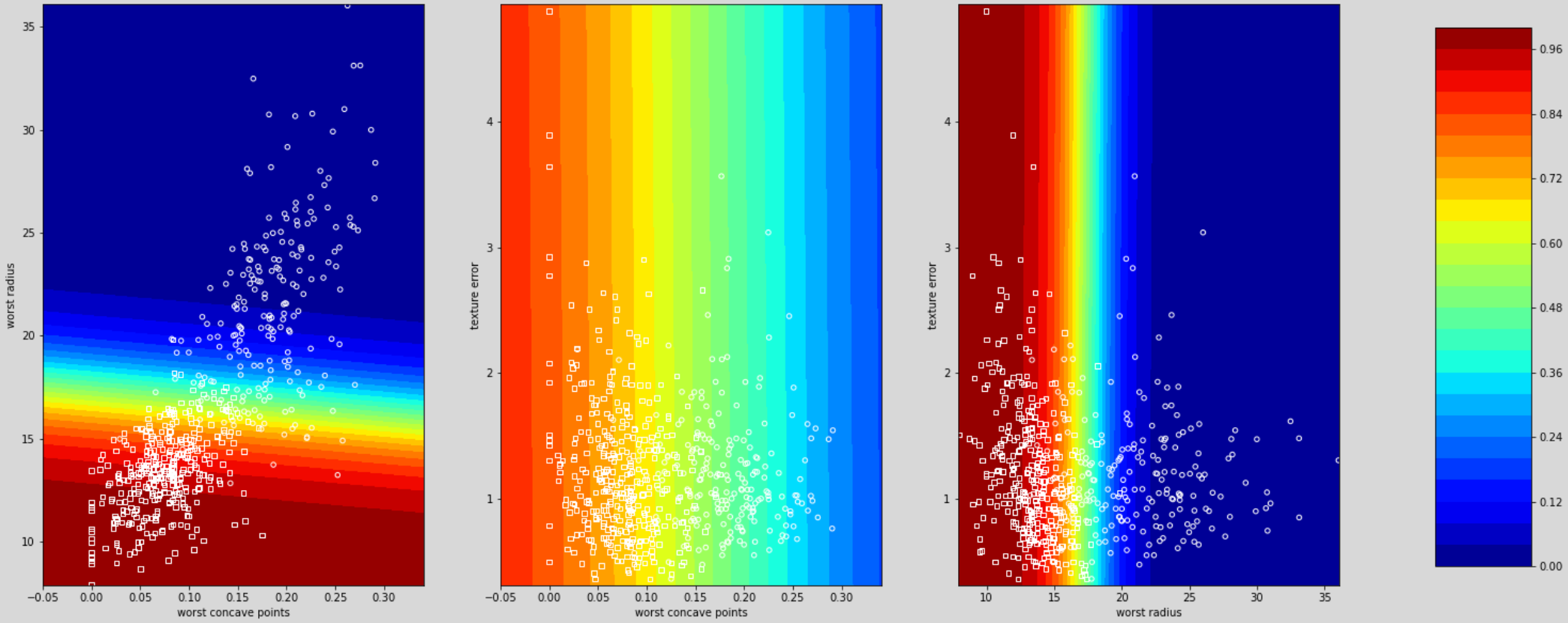




Dataset 2: Breast Cancer Dataset (Logistic Regression)

There are 30 features in the dataset, but I only chose (1) worst radius, (2) worst concave points and (3) texture error. Note that by selecting these features, we can see that there is good linear separation between data points.

Eta = 0.01, epochs = 300

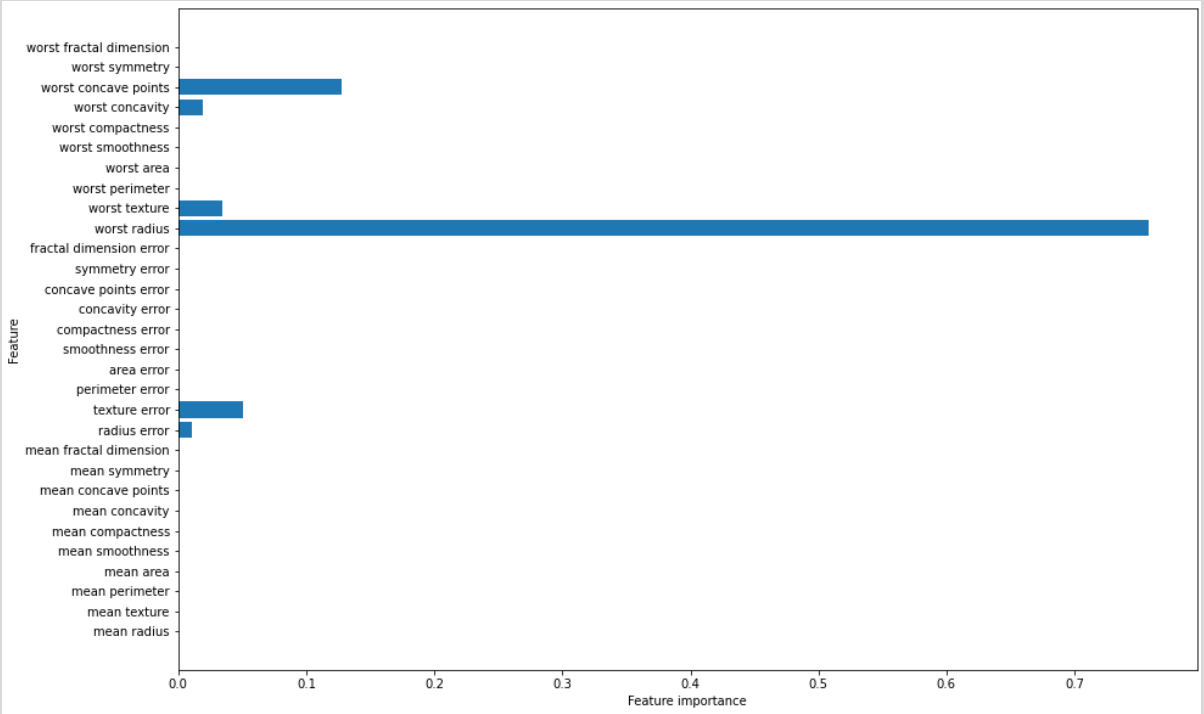


Dataset 2: Breast Cancer Dataset

To decide on the choice of features, I had to use an ML technique outside the scope of this course (feature importance by decision tree classifier). Nevertheless, I present it here for completeness.

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(max_depth =3, random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test,
y_test)))
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_,
align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
plt.figure(figsize=(15,10))
plot_feature_importances_cancer(tree)
```



Dataset 3: Iris Dataset

There are 150 data points, 50 for each type of iris (setosa, versicolor, virginica). There are four features: sepal length, sepal width, petal length, and petal width.

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Shape of iris data: {}".format(iris.data.shape))
print("Sample counts per class:\n{}".format( {n: v for n, v in zip(iris.target_names,
np.bincount(iris.target))}))
print("Feature names:\n{}".format(iris.feature_names))
```

Output:

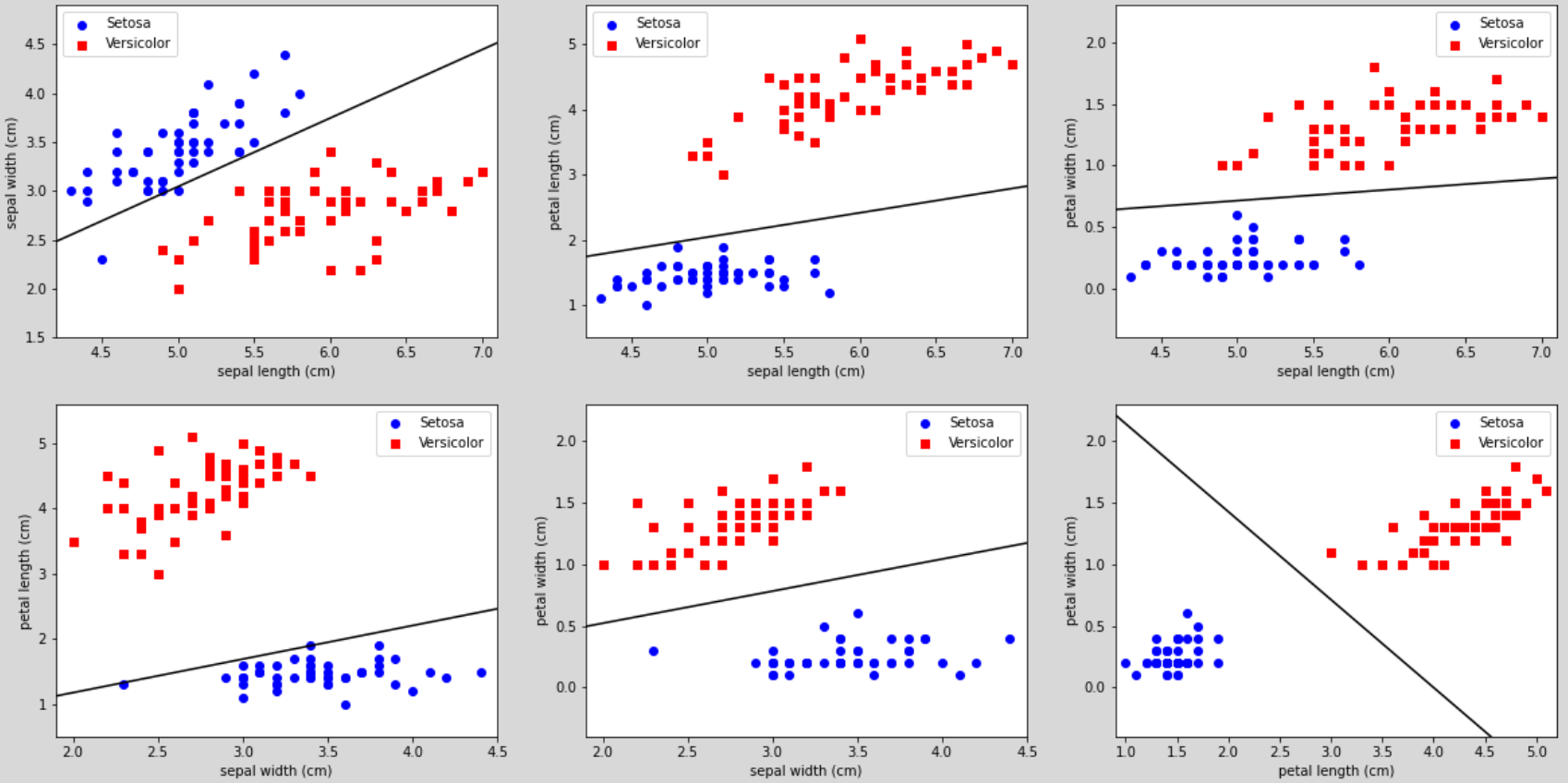
```
Shape of cancer data: (150, 4)
Sample counts per class: {'setosa': 50, 'versicolor': 50, 'virginica': 50}
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width
(cm)']
```

In this example, I only classify setosa and versicolor.

Dataset 3: Iris Dataset(Perceptron)

Note that there is good linear separation between the datapoints, so the binary classifier works quite well.

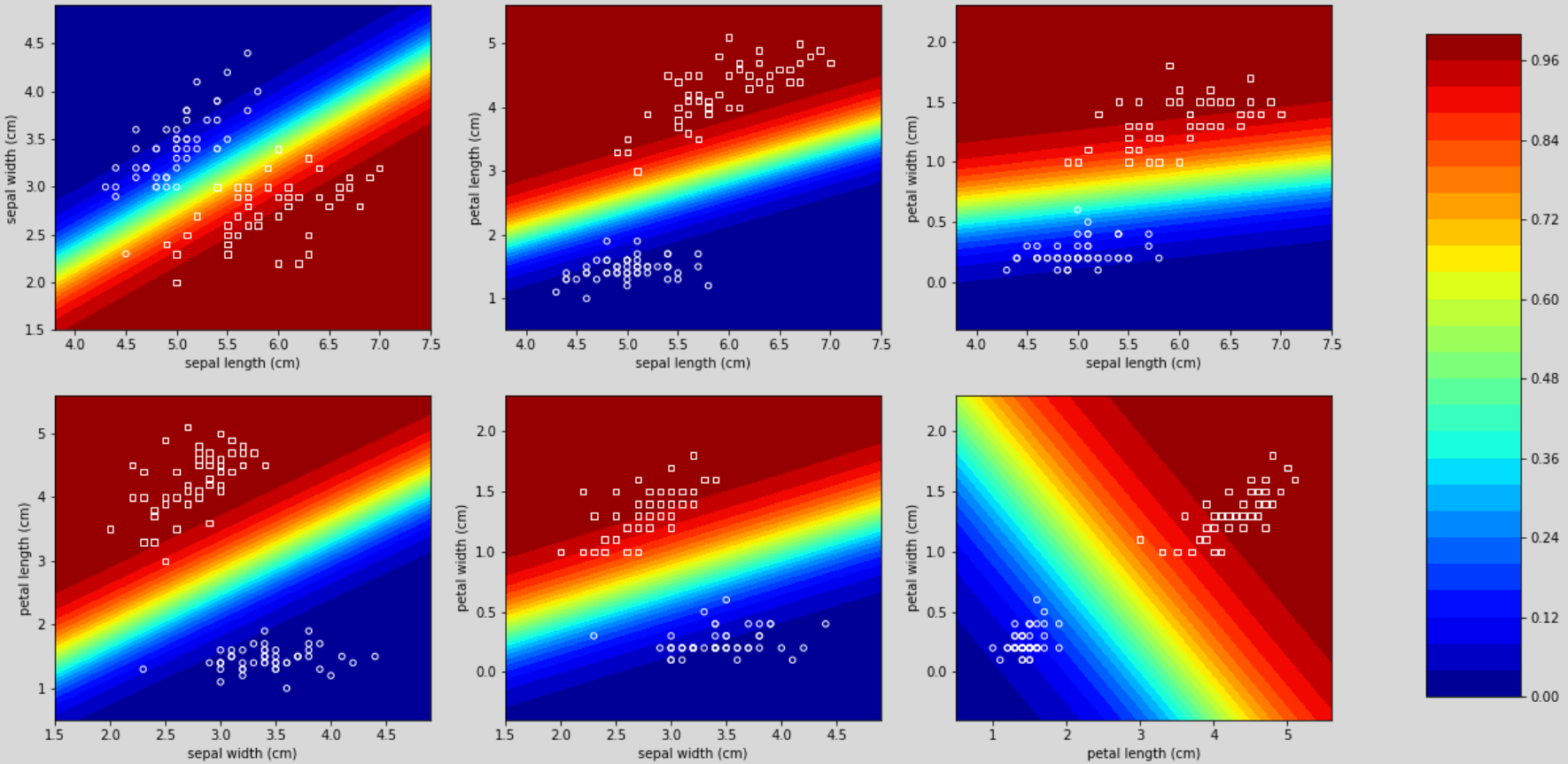
Eta = 0.05, epochs = 100



Dataset 3: Iris Dataset(Perceptron)

Note that there is good linear separation between the datapoints, so the binary classifier works quite well.

Eta = 0.05, epochs = 100



Self-Reflection

- I liked this activity since it allows us to dip our toes into machine learning by simple binary classification. The algorithm is also quite easy to follow and code, and it is satisfying to see it put into action in real-life datasets.

Self-score: 95/100

References

Soriano, M. (2020), Applied Physics 157 Module, "*ML2 – Perceptron*"

Müller, A. C., & Guido, S. (2018). *Introduction to machine learning with python: A guide for data scientists*. O'Reilly Media.