Course: CSCE 5215 Machine Learning

Professor: Zeenat Tariq

Activity 6

Implementation of Linear Regression from Scratch

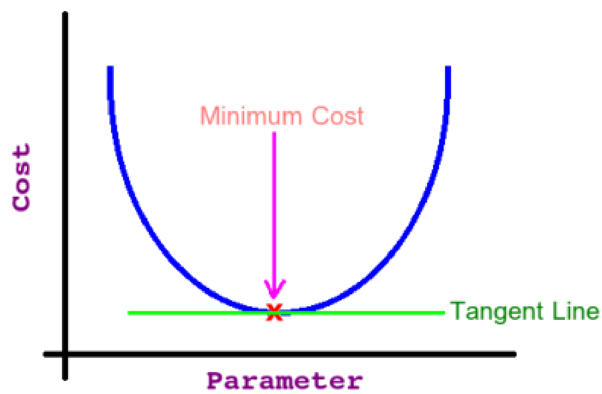
Linear regression is a supervised learning algorithm used for regression tasks, where the goal is to predict a continuous output variable based on one or more input features.

In [76]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_breast_cancer, load_diabetes, load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, accuracy_score, precision_score, recall_score
from IPython.display import Image
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, accuracy_score, precision_score, recall_score
```

One way to think about gradient descent is. that you start at some arbitrary point on the surface, look to see in which direction is the "hill"

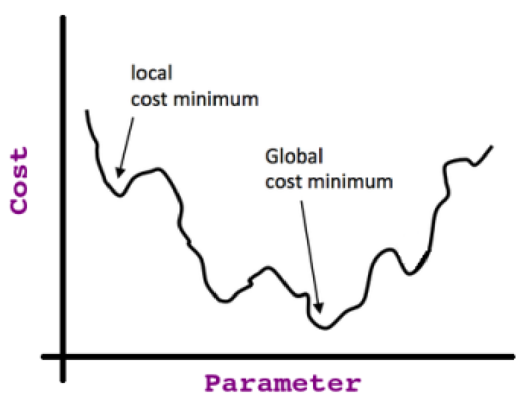
In [77]:
```
Image(url="https://i.stack.imgur.com/mZ9UU.png")
```

Out[77]:



In [78]:
```
Image(url="https://i.stack.imgur.com/WYEux.png")
```

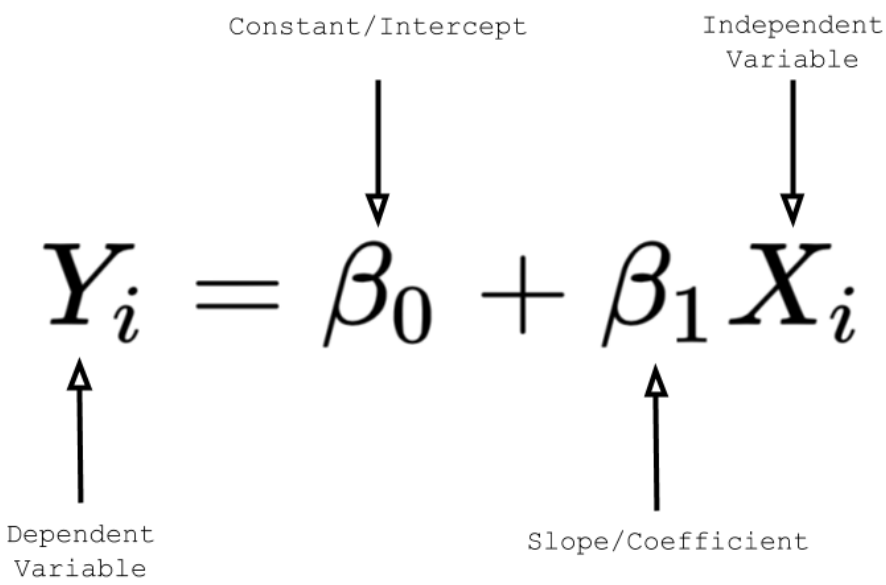
Out[78]:



Implimenting Linear regression from scratch

In [79]:
```
Image(url="https://miro.medium.com/v2/resize:fit:1400/1*GSAcN9G7stUJQbuOhu0HEg.png")
```

Out[79]:



LinearRegression fits a linear model with coefficients w = (w1, ..., wp) to minimize the residual sum of squares between the observed targets in the dataset

In [80]:
```
class LinearRegression:
    def __init__(self, lr=0.001, n_iters=10):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.random.random(n_features)
        self.bias = np.random.random(1)

        for _ in range(self.n_iters):
            y_pred = np.dot(X, self.weights) + self.bias

            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            self.weights = self.weights - self.lr * dw
            self.bias = self.bias - self.lr * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias
```

Loading the data

In [81]:
```
data = load_diabetes()
X =data.data
y=data.target
```

Use train_test_split function to split the dataset into training and testing sets. The training set consists of 80% of the data, and the testing set contains the remaining 20%. The random seed of 1234 ensures reproducibility of the split.

In [82]:
```
# Splitting the dataset into training and testing sets
x_train, x_test, y_train, y_test  = train_test_split(X, y, test_size=0.2, random_state=1234)
```

An instance of the LinearRegression class is created with a learning rate of 0.01. The model is trained on the training data (X_train and y_train) using the fit method.

In [83]:
```
# Initializing and training a linear regression model
reg = LinearRegression(lr=0.01)
reg.fit(x_train, y_train)
```

The trained model is used to make predictions on the test set (X_test), and store the predicted values in predictions.

In [84]:
```
# Making predictions on the test set
predictions = reg.predict(x_test)
predictions
```

Out[84]:
```
array([15.35824968, 15.24932774, 15.03356907, 14.99895647, 14.99213402,
       15.2310342 , 15.31979069, 15.16715319, 15.10186332, 15.26168991,
       14.97493837, 15.34209939, 15.17421797, 15.24741232, 15.14001902,
       15.21134736, 15.0908676 , 15.131002  , 15.50544235, 15.28105629,
       15.25054458, 15.10714634, 15.06967532, 15.37689065, 15.34037206,
       15.40583029, 15.03570169, 15.47271306, 15.31683936, 14.94171465,
       15.08390584, 15.13486152, 15.08642702, 15.00646486, 15.23787692,
       15.39108233, 15.40586858, 14.94371057, 15.35166162, 15.04858218,
       15.24330151, 15.14004651, 15.38823207, 15.10361423, 15.23820081,
       15.10045882, 15.11511418, 15.30184251, 15.10255239, 15.27679468,
       15.13567322, 15.19911419, 15.32329269, 15.0939519 , 15.05757925,
       15.23413674, 15.30438661, 15.19007501, 15.3829834 , 15.18543132,
       14.98107649, 15.35239867, 15.26855864, 15.17915789, 15.27473192,
       15.45773548, 15.30248322, 15.29789821, 15.21344117, 15.34336036,
       14.93137693, 15.21339762, 15.2949479 , 15.05019388, 15.24029888,
       15.3087602 , 15.12698734, 15.28831589, 15.28671247, 15.44516265,
       15.22351843, 14.96342519, 15.11932541, 15.24835497, 15.04860691,
       15.04446235, 15.34007878, 15.11430787, 15.38934183])
```

Define a function called mse that calculate the mean squared error (MSE) between the true target values (y_test) and the predicted values (predictions).

In [85]:
```
# Defining a function to calculate mean squared error (MSE)
def mse(y_test, predictions):
    return np.mean((y_test - predictions) ** 2)
```

In [86]:
```
print(mse(y_test, predictions))
```

```
24053.462129123734
```

Implementation of Logistic Regression from Scratch

A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point.

In [87]:
```
Image(url="https://qph.cf2.quoracdn.net/main-qimg-0c921e324b298fdc72027d25ee584db3.webp")
```

Out[87]:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Softmax is an activation function that scales numbers/logits into probabilities.

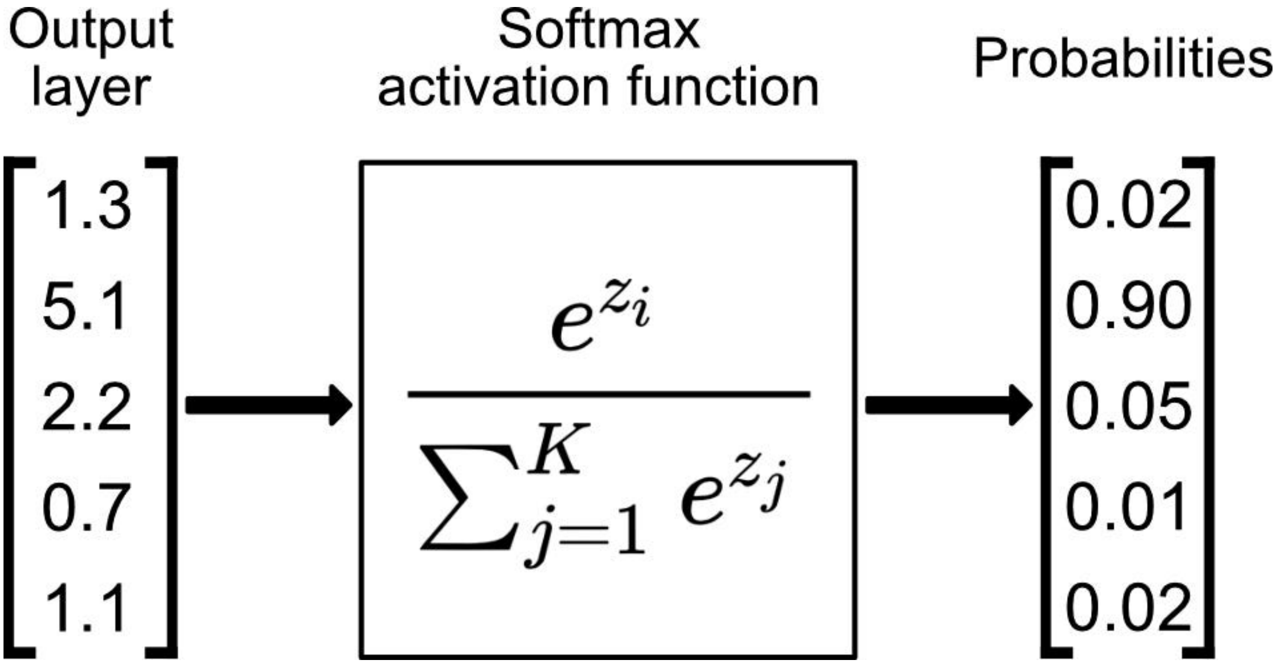
In [88]:
```
Image(url="https://miro.medium.com/v2/resize:fit:728/1*ui7n5s48-qNF7BBGfDPioQ.png")
```

Out[88]:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

In [89]:
```
Image(url="https://miro.medium.com/v2/resize:fit:1400/1*ReYpdIZ3ZSAPb2W8cJpkBg.jpeg")
```

Out[89]:



Logistic regression is a data analysis technique that uses mathematics to find the relationships between two data factors

In [90]:

```
class LogisticRegression:
    def __init__(self, learning_rate=0.001, n_iters=10, model=None, thr=0.5):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
        self.model = model
        self.thr = thr

    def fit(self, X, y):
        n_samples, n_features = X.shape

        if self.model == 'sigmoid':
            self.weights = np.random.random(n_features)
            self.bias = np.random.random(1)
        elif self.model == 'softmax':
            num_classes = len(np.unique(y))
            self.weights = np.random.random((n_features, num_classes))
            self.bias = np.random.random((1, num_classes))

        for _ in range(self.n_iters):
            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = self.select_activation(linear_model)

            if self.model == 'sigmoid':
                y_predicted_cls = [1 if i > self.thr  else 0 for i in y_predicted]
                y_diff = y_predicted - y
            elif self.model == 'softmax':
                y_one_hot = np.eye(num_classes)[y]
                y_diff = y_predicted - y_one_hot

            # compute gradients
            dw = (1 / n_samples) * np.dot(X.T, y_diff)
            db = (1 / n_samples) * np.sum(y_diff, axis=0)
            # update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.select_activation(linear_model)
        if self.model == 'sigmoid':
            y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
        elif self.model == 'softmax':
            y_predicted_cls = np.argmax(y_predicted, axis=1)
        return np.array(y_predicted_cls)

    def select_activation(self, x):
      if self.model =="sigmoid":
```

```
            return 1 / (1 + np.exp(-x))
        else:
            exp_z = np.exp(x - np.max(x, axis=1, keepdims=True))
            return exp_z / exp_z.sum(axis=1, keepdims=True)
```

Load data and split the dataset

In [91]:
```
data = load_breast_cancer()
X = data.data
y = data.target
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42, stratify=y)
```

load the model

In [92]:
```
log_reg = LogisticRegression(model="sigmoid")
model = log_reg.fit(x_train, y_train)
y_pred = log_reg.predict(x_test)
```

```
<ipython-input-90-8cfad41c3e4a>:50: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-x))
```

Compute F1 score, precision, and recall

In [93]:
```
Image(url="https://assets-global.website-files.com/5d7b77b063a9066d83e1209c/639c3cc56bda8713d4a2f29c_precisi
```

Out[93]:

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

In [94]:
```
Image(url="https://assets-global.website-files.com/5d7b77b063a9066d83e1209c/639c3d2a22f93657640ef19f_f1-scor
```

Out [94]:

$$\text{F1 Score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

In [95]:
```
y_pred_train = log_reg.predict(x_train)
print(f"accuracy_train: {accuracy_score(y_train, y_pred_train)}")
print(f"f1_score_train: {f1_score(y_train, y_pred_train)}")
print(f"precision_train: {precision_score(y_train, y_pred_train)}")
print(f"recall_train: {recall_score(y_train, y_pred_train)}\n")

print(f"accuracy_test: {accuracy_score(y_test, y_pred)}")
print(f"f1_score_test: {f1_score(y_test, y_pred)}")
print(f"precision_test: {precision_score(y_test, y_pred)}")
print(f"recall_test: {recall_score(y_test, y_pred)}")
```

```
accuracy_train: 0.640625
f1_score_train: 0.6275303643724696
precision_train: 0.8959537572254336
recall_train: 0.48286604361370716

accuracy_test: 0.6666666666666666
f1_score_test: 0.6545454545454545
precision_test: 0.9473684210526315
recall_test: 0.5
```

Apply softmax

Load the data and split

In [96]:
```
data = load_iris()
X = data.data
y = data.target
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42, stratify=y)
```

load the model

In [97]:
```
log_reg = LogisticRegression(model="softmax")
model = log_reg.fit(x_train, y_train)
y_pred = log_reg.predict(x_test)
```

Performance result

In [98]:
```
y_pred_train = log_reg.predict(x_train)
print(f"accuracy_train: {accuracy_score(y_train, y_pred_train)}")

print(f"accuracy_test: {accuracy_score(y_test, y_pred)}")
```

```
accuracy_train: 0.3333333333333333
accuracy_test: 0.3333333333333333
```

Let's use sklearn to hypertune Logistic Regression

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model

https://scikit-learn.org/stable/modules/linear_model.html

In [99]:
```
Image(url="https://miro.medium.com/v2/resize:fit:2000/1*zMLv7EHYtjfr94JOBzjqTA.png")
```

Out[99]:

L1 regularization on least squares:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \boxed{\lambda \sum_{i=1}^{k} |w_i|}$$

L2 regularization on least squares:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \boxed{\lambda \sum_{i=1}^{k} w_i^2}$$

In [117…
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
log_reg = LogisticRegression()

param_grid = {
    "solver": ["liblinear", "sag", "saga"],
    "penalty": ["l1", "l2"],
    "class_weight": ["balanced"],
    "C": [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100],
}
grid_regression = GridSearchCV(log_reg, param_grid, scoring="accuracy",cv=5)
# grid_regression.fit(x_train, y_train)
```

In [101…
```
y_pred = grid_regression.best_estimator_.predict(x_test)
accuracy_score(y_pred, y_test)
```

Out[101]: 1.0

In [102…
```
import sklearn
sklearn.metrics.get_scorer_names()
```

Out[102]:
```
['accuracy',
 'adjusted_mutual_info_score',
 'adjusted_rand_score',
 'average_precision',
 'balanced_accuracy',
 'completeness_score',
 'explained_variance',
 'f1',
 'f1_macro',
 'f1_micro',
 'f1_samples',
 'f1_weighted',
 'fowlkes_mallows_score',
 'homogeneity_score',
 'jaccard',
 'jaccard_macro',
 'jaccard_micro',
 'jaccard_samples',
 'jaccard_weighted',
 'matthews_corrcoef',
 'max_error',
 'mutual_info_score',
 'neg_brier_score',
 'neg_log_loss',
 'neg_mean_absolute_error',
 'neg_mean_absolute_percentage_error',
 'neg_mean_gamma_deviance',
 'neg_mean_poisson_deviance',
 'neg_mean_squared_error',
 'neg_mean_squared_log_error',
 'neg_median_absolute_error',
 'neg_negative_likelihood_ratio',
 'neg_root_mean_squared_error',
 'normalized_mutual_info_score',
 'positive_likelihood_ratio',
 'precision',
 'precision_macro',
 'precision_micro',
 'precision_samples',
 'precision_weighted',
 'r2',
 'rand_score',
 'recall',
 'recall_macro',
 'recall_micro',
 'recall_samples',
 'recall_weighted',
 'roc_auc',
 'roc_auc_ovo',
 'roc_auc_ovo_weighted',
 'roc_auc_ovr',
 'roc_auc_ovr_weighted',
 'top_k_accuracy',
 'v_measure_score']
```

Practice

In [103…
```
# Import libraries
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import pandas as pd
```

In [104…
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive",
force_remount=True).
```

In [105…
```
# Load the data
file_path = '/content/drive/MyDrive/ColabNotebooks/Pumpkin_Seeds_Dataset.xlsx'

data = pd.read_excel(file_path)
# make sure you also provide the kaggle link here
# https://www.kaggle.com/datasets/muratkokludataset/pumpkin-seeds-dataset

data['Class']=data['Class'].replace({'Çerçevelik':1,'Ürgüp Sivrisi':0})
X = data.iloc[:,:-1]
y = data['Class']
feature_names = data.columns
```

```
print(feature_names)
data.head()
```

```
Index(['Area', 'Perimeter', 'Major_Axis_Length', 'Minor_Axis_Length',
       'Convex_Area', 'Equiv_Diameter', 'Eccentricity', 'Solidity', 'Extent',
       'Roundness', 'Aspect_Ration', 'Compactness', 'Class'],
      dtype='object')
```

Out[105]:

	Area	Perimeter	Major_Axis_Length	Minor_Axis_Length	Convex_Area	Equiv_Diameter	Eccentricity	Solidity	Extent	Roundne
0	56276	888.242	326.1485	220.2388	56831	267.6805	0.7376	0.9902	0.7453	0.89
1	76631	1068.146	417.1932	234.2289	77280	312.3614	0.8275	0.9916	0.7151	0.84
2	71623	1082.987	435.8328	211.0457	72663	301.9822	0.8749	0.9857	0.7400	0.76
3	66458	992.051	381.5638	222.5322	67118	290.8899	0.8123	0.9902	0.7396	0.84
4	66107	998.146	383.8883	220.4545	67117	290.1207	0.8187	0.9850	0.6752	0.83

In [106…

```
# Split the dataset into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```

In [107…

```
# Initialize and training a logistic regression model
logistic_regression = LogisticRegression()
logistic_regression.fit(X_train, y_train)
```

Out[107]:

```
▾ LogisticRegression
LogisticRegression()
```

In [108…

```
# Make predictions on the test set
pred = logistic_regression.predict(X_test)
```

In [109…

```
# Calculate f1_score, accuracy_score, precision_score, recall_score

f1_score_value = f1_score(y_test, pred)
accuracy_score_value = accuracy_score(y_test, pred)
precision_score_value = precision_score(y_test, pred)
recall_score_value = recall_score(y_test, pred)

#print all the values
print(f"F1 Score: {f1_score_value}")
print(f"Accuracy: {accuracy_score_value}")
print(f"Precision: {precision_score_value}")
print(f"Recall: {recall_score_value}")
```

```
F1 Score: 0.8733850129198966
Accuracy: 0.8693333333333333
Precision: 0.8802083333333334
Recall: 0.8666666666666667
```

In [110…

```
# Tune Logistic Regression using GridSearchCV, make sure you play with these hyperparameters: penalty, C, so
# https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

param_grid = {
   'C' : [0.1, 0.5, 1],
   'solver': [ 'sag', 'saga', 'liblinear'],
   'max_iter': [100, 150],
   'penalty' : ['l2'],
   'class_weight': ['balanced'],
   'random_state' : [42]
}

# Create a Logistic Regression model
logistic_regression = LogisticRegression()
```

In [111…

```
# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=logistic_regression,param_grid=param_grid,scoring='accuracy',cv=5,n_job
```

In [112…

```
# Fit the grid search to your training data
grid_search.fit(X_train, y_train)
```

Out[112]:

```
▸ GridSearchCV
▸ estimator: LogisticRegression
  ▸ LogisticRegression
```

In [113…

```
# Get the best hyperparameters and model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
```

In [114…

```
# Use the best model for predictions
y_pred_best = best_model.predict(X_test)
```

In [115…

```
# Evaluate the best model
f1_score_best = f1_score(y_test, y_pred_best)
accuracy_best = accuracy_score(y_test, y_pred_best)
precision_best = precision_score(y_test, y_pred_best)
recall_best = recall_score(y_test, y_pred_best)

print("Best Hyperparameters:")
print(best_params)

print("Evaluation Metrics for the Best Model:")
print(f"F1 Score: {f1_score_best}")
print(f"Accuracy: {accuracy_best}")
print(f"Precision: {precision_best}")
print(f"Recall: {recall_best}")
```

```
Best Hyperparameters:
{'C': 0.1, 'class_weight': 'balanced', 'max_iter': 100, 'penalty': 'l2', 'random_state': 42, 'solver': 'libl
inear'}
Evaluation Metrics for the Best Model:
F1 Score: 0.8709256844850065
Accuracy: 0.868
Precision: 0.8859416445623343
Recall: 0.8564102564102564
```

In [116…

```
# get the LogisticRegression that we implemented from scratch and modify learning_rate, and n_iters, providi

import numpy as np
from sklearn.model_selection import train_test_split

# Split your data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

learning_rates = [0.001, 0.01, 0.1]
n_iters_values = [10, 20, 30]

best_model = None
best_learning_rate = None
best_n_iters = None
best_accuracy = 0

for learning_rate in learning_rates:
    for n_iters in n_iters_values:
        log_reg = LogisticRegression()
        log_reg.lr = learning_rate
        log_reg.n_iters = n_iters

        log_reg.fit(X_train, y_train)
        y_pred = log_reg.predict(X_test)

        # Calculate accuracy
        accuracy = (y_pred == y_test).mean()

        if accuracy > best_accuracy:
            best_model = log_reg
            best_learning_rate = learning_rate
            best_n_iters = n_iters
            best_accuracy = accuracy

print("Best Learning Rate:", best_learning_rate)
print("Best n_iters:", best_n_iters)
print("Best Accuracy:", best_accuracy)
```

```
Best Learning Rate: 0.001
Best n_iters: 10
Best Accuracy: 0.86
```

Write your understanding of the model in 200 to 400 words

I loaded the pumpkin seed dataset containing features like Area, Perimeter, Class, etc. The target feature is Class. There are 2 values in class - 'Çerçevelik', 'Ürgüp Sivrisi'. First I replace 'Çerçevelik' with 1 and 'Ürgüp Sivrisi' with 0.

After splitting the data into test and train (test size is 30% and train size 70%), use scikit-learn logistic regression model for binary classification. I trained the model on the training data and evaluated its performance using common classification metrics like F1-score, accuracy, precision, and recall. The results indicated an F1-score of 0.9, an accuracy of 0.896, a precision of 0.9, and a recall of 0.9.

An F1-score of 0.9 indicates the model's strong overall performance. An accuracy of 0.896 means it's correct 89.6% of the time. A precision of 0.9 suggests that when it predicts a positive, it's right 90% of the time. A recall of 0.9 means it correctly identifies 90% of actual positives.

To improve the logistic regression model's performance, hyperparameter tuning is done. GridSearchCV is a powerful tool in sci-kit-learn that automates the search for the the best combination of hyperparameters. I defined a set of hyperparameters, including 'C,' 'solver,' 'max_iter,' 'penalty,' and 'class_weight.' GridSearchCV helped to identify the best hyperparameters.

The "Best Hyperparameters" are the configuration settings that produced the highest-performing logistic regression model during hyperparameter tuning. These settings are as follows:

C: 0.1 Class Weight: 'balanced' Max Iterations: 100 Penalty: 'l2' Random State: 42 Solver: 'liblinear'

Using the LogisticRegression that is defines I gave 3 values for learining_rate and n_iters_values. Then I looped it with all the possible combinations to get the best values (Best Learning Rate, Best n_iters, Best Accuracy) Among those the below are the best values.

Best Learning Rate: 0.001 Best n_iters: 10 Best Accuracy: 0.86