به نام خدا

ML4022_MP1

دانشجو : مصطفی نبی پور

شماره دانشجویی : 40112864

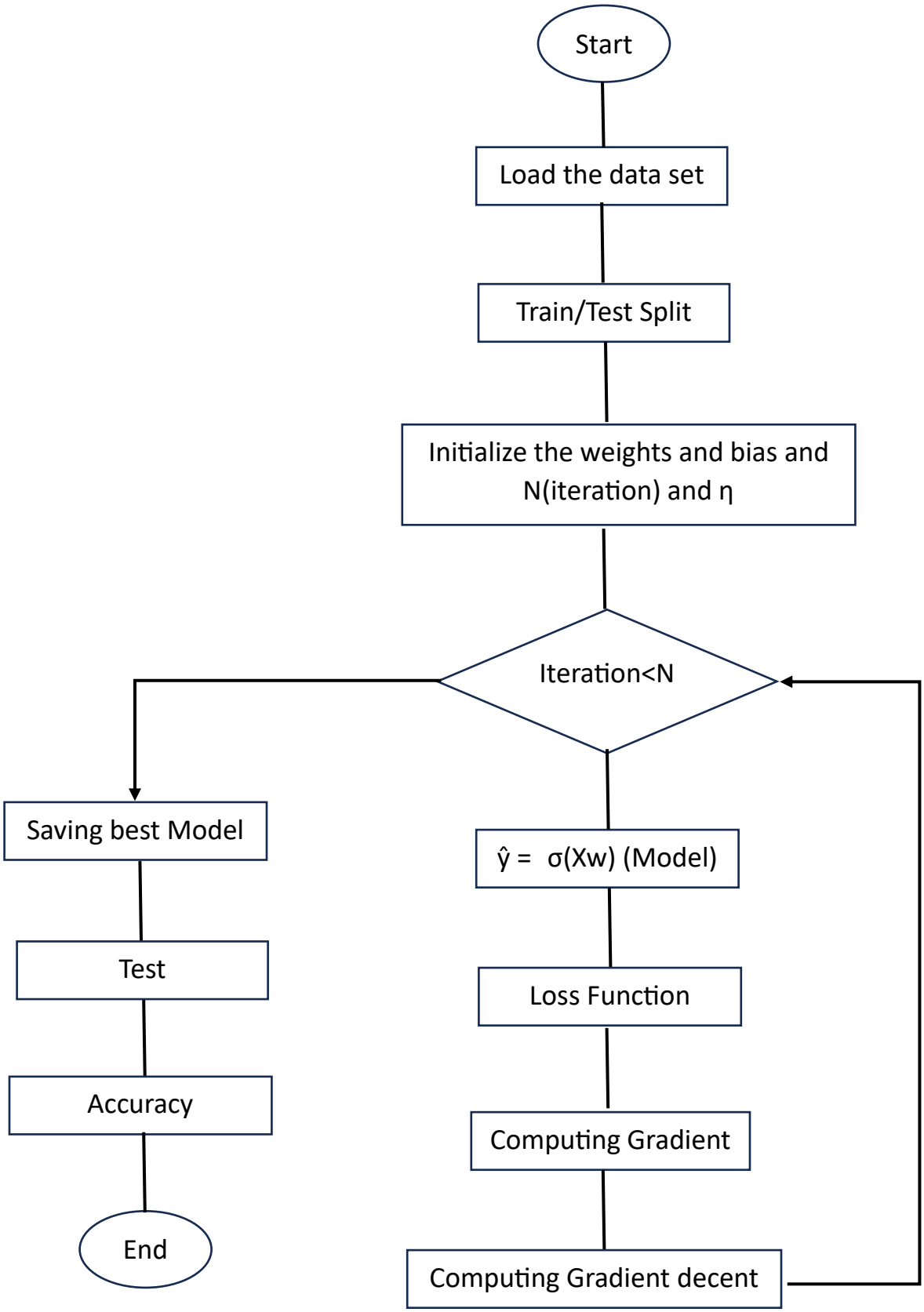
لینک github :

https://github.com/mostafanb77/ML4022_MP1

لینک google drive :

https://drive.google.com/drive/folders/15pi8D_WDpbI0eHCEs9wxz255d6VomYfq?usp=sharing

سوال اول (۱)


Start
Load the data set
Train/Test Split
Initialize the weights and bias and N(iteration) and η
Iteration<N
Saving best Model
Test
Accuracy
End
ŷ = σ(Xw) (Model)
Loss Function
Computing Gradient
Computing Gradient decent

در وهله اول باید داده ها رو وارد کنیم سپس باید آنها رو به ۲ دسته Train-Test تقسیم بندی کنیم.سپس باید مدل خود را درست کنیم (به طور مثال Logistic regression) بعد از آن باید وزن ها (W0,W1,…,Wn) را درست کرده و ضریب گرادیان (η) و مقدار تکرار مدل برای (N) برای پیدا کردن بهترین مدل را معلوم کنیم.

سپس داده های Train را وارد مدل کرده و پس از آن آنرا وارد Loss Function (در اینجا به عنوان مثال BCE) میکنیم سپس گرادین وزن های آنرا حساب کرده ($\nabla_w L(w) = \frac{1}{n} X^T(\hat{y} - y)$) بعد از آن از روش Gradian decsent ($w = w - \eta \nabla_w L(w)$) وزن ها رو بروز میکنیم.سپس تا N تعداد مدل را اجرا کرده سپس بهترین مدل را ذخیره میکنیم و آنرا بر روی داده های Test امتحان میکنیم و در آخر Accuracy مدل خود را بدست می آوریم.

اگر بخواهیم از ۲ کلاسه به چند کلاسه طبقه بندی ما تغییر کند باید Model و Loss Function ما تغییر پیدا کند ولی بقیه قسمت ها سر جای خود هستند. در واقع دلیل تغییر این ۲ این است که مثلا در Logistic Regression هدف ما این است که ۲ کلاس (0,1) را از هم جدا کنیم و اگر ما چند کلاس داشته باشیم این مدل دیگر به کار ما نمی آید .

همچنین Loss Function ما نیز به طور مثال در اینجا Binary Cross Entropy می باشد که فرمول آن بدین شکل است : $-\frac{1}{n}(y^T \log(\hat{y}) + (1-y)^T \log(1-\hat{y}))$

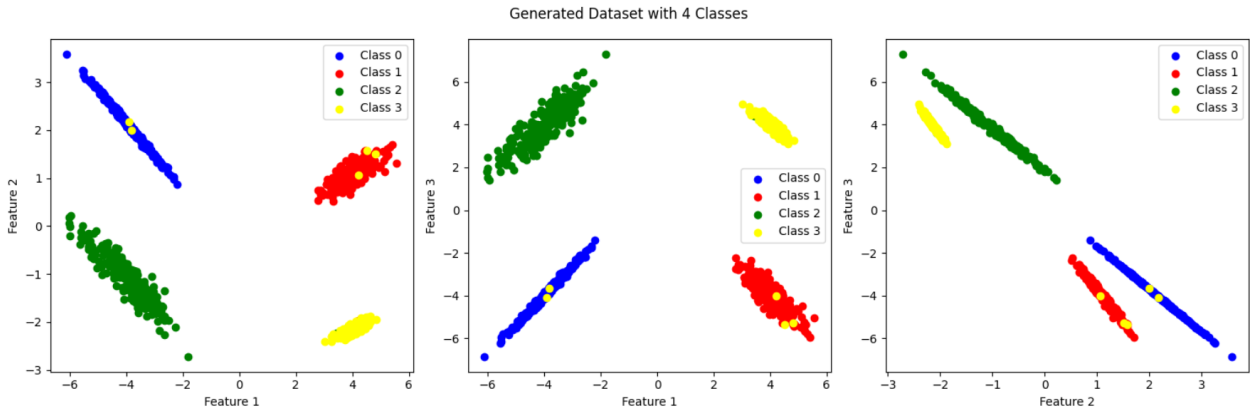
همینطور که میبینید این تابع در واقع برای طبقه بندی ۲ کلاسه میباشد پس باید Loss Function نیز تغییر کند.

۲)

دیتاست را ما بدین شکل بدست آوردیم :

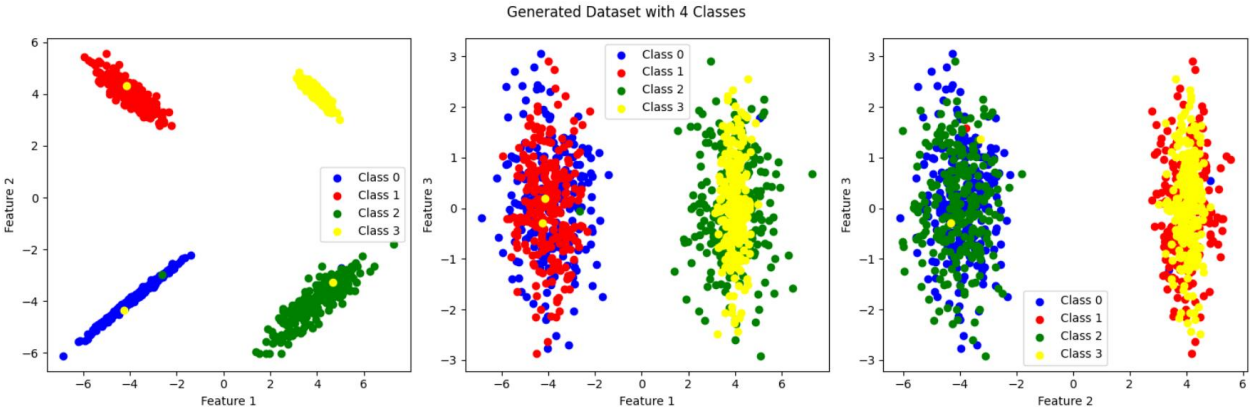
```
X, y = make_classification(n_samples=1000, n_redundant=1,
n_clusters_per_class=1, class_sep=4, n_features=3, n_classes=4,
random_state=64)
```

اگر دیتاست خود را ۲ به ۲ نمایش دهیم بدین شکل میشود (مثلا feature1&2 , ...)



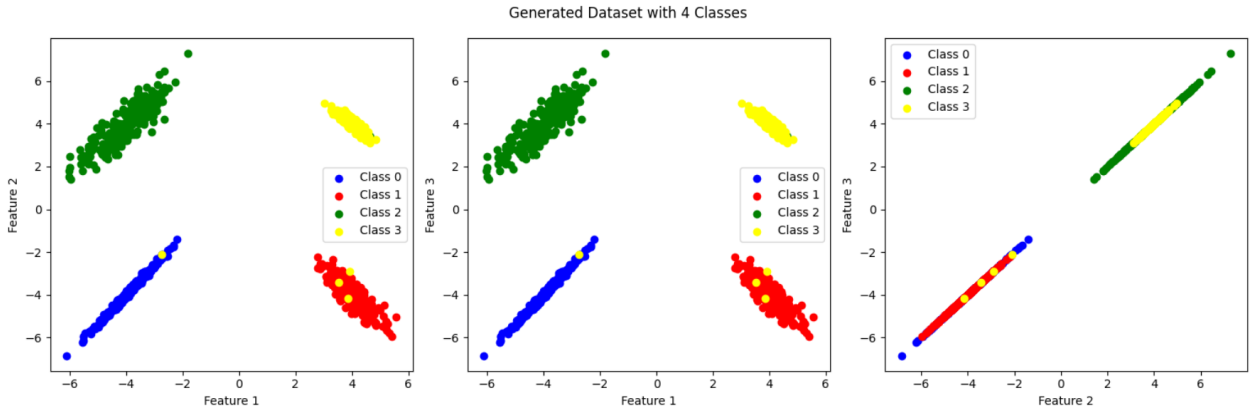
همینطور که میبنید کلاس ها را راحت تر میتوان جدا کرد(پس زیاد چالش بر انگیز نیست) ولی اگر در حالتی که *n_redundant = 0* بخوایم *make classification* را انجام بدهیم کلاس ها *overlap* زیادی دارند. در واقع موضوعی که این جا مهم بود اضافه کردن یک کلاس *redundant* بود تا داده ها راحت تر از هم جدا شوند.

برای اینکه داده ها را سخت تر از هم جدا کنیم میتوان *redundant class* را حذف کرد که بدین شکل میشود:



همینطور که مشاهده میکنید تنها فقط مجموع *1,2* را میتوان راحت تر جدا کرد ولی بقیه نویز زیادی دارند.

کار دیگری که میتوان انجام داد اضافه کردن *n_repeated* (برابر با ۱) و حذف *n_redundant* میباشد که این تفکیک کلاس ها رو در آخرین مورد سخت میکنه :



داده های زرد رنگ موجود در بقیه کلاس ها داده ای نویز دار هستند که برای ما اهمیت زیادی ندارند.

کارهایی که باعث سخت تر شدنه طبقه بندی داده ها میشن :

1)*Class_sep* رو کم کنیم تا فاصله کلاس ها را کم کند.

2)افزایش *flip_y* باعث زیاد شدن نویز میشه و احتمال اینکه داده ها به صورت نادرست کلاس بندی بشن رو افزایش میده.

3)افزودن *weights* به داده ها باعث میشه که عدم تعادل داده های در هر کلاس بوجود بیاد.

4)افزودن *n_redundant* باعث میشه که *feature* ها به هم وابستگی داشته باشند و در این صورت *overfitting* اتفاق بیفتد.

گزارش کد :

خوب در وهله اول کتابخانه های مورد نیاز را وارد میکنیم :

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
import numpy as np
```

این دستورات کتابخانه های مورد نیاز میباشند. اولی برای پلات کردن دومی برای درست کردن داده و آخری برای کار های محاسباتی میباشد.

```
X, y = make_classification(n_samples=1000, n_redundant=1,
n_clusters_per_class=1, class_sep=4, n_features=3, n_classes=4,
random_state=64)
```

این هم برای درست کردن داده میباشد که اولین ویژگی آن تعداد داده ها دومی تعداد *feature* وابسته در واقع تعداد *feature* مستقل برابر ۲ است(*n_informative*) و *1 feature* ترکیب خطی ۲ ویژگی مستقل میباشد.

بعدی تعداد خوشه ها در هر کلاس را معلوم میکند.بعدی ***class_sep*** در واقع کلاس ها رو از هم جدا میکند.

بعد از آن تعداد ***feature*** ها سپس تعداد کلاس ها و در آخر ***random_state*** میباشد که معلوم کردن آن برای این است که هر بار که کد اجرا شد به یک شکل داده ها جدا بشوند.

```
colors = np.array(['blue', 'red', 'green', 'yellow'])

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

feature_pairs = [(0, 1), (0, 2), (1, 2)]

for i, ax in enumerate(axs):
    for j in range(4):
        ax.scatter(X[y == j, feature_pairs[i][0]], X[y == j,
feature_pairs[i][1]], label=f'Class {j}', c=colors[j])
    ax.set_xlabel(f'Feature {feature_pairs[i][0]+1}')
    ax.set_ylabel(f'Feature {feature_pairs[i][1]+1}')
    ax.legend()

plt.suptitle('Generated Dataset with 4 Classes')
plt.tight_layout()
plt.show()
```

خط اول این بخش آرایه ای از رنگ ها را ایجاد می کند که در آن هر رنگ با برچسب کلاس متفاوتی مطابقت دارد. در این مورد، 4 کلاس وجود دارد، بنابراین 4 رنگ وجود دارد.

خط بعدی یک شکل با سه ***subplot*** که به صورت افقی مرتب شده اند ایجاد می کند که هر کدام یک جفت ***Feature*** را نشان می دهد.

خط بعدی جفت ***Feature*** هایی را که در هر طرح فرعی ترسیم می شود را مشخص می کند. به عنوان مثال،

(0، 1) نشان دهنده جفت ***Feature*** های 1 و 2 است.

در قسمت بعد کد بر روی هر ***subplot***(***ax***) در آرایه ***axs*** تکرار می‌شود، در حالی که با استفاده از ***enumerate***، ***index*** را نیز پیگیری می‌کند. تابع ***enumerate(axs)*** جفت‌های (***index,value***) را برمی‌گرداند، جایی که ***i*** نمایانگر ***subplot index*** و ***ax*** نشان‌دهنده خود ***subplot*** است.

در داخل حلقه بیرونی، یک حلقه دیگر وجود دارد که در محدوده ای از مقادیر از 0 تا 3 (***range(4)***) تکرار می شود. این حلقه روی هر برچسب کلاس (***j***) از 0 تا 3 تکرار می شود.

در حلقه تو در تو، برای هر برچسب کلاس (*j*) یک نمودار *scatter* بر روی *subplot* فعلی (*ax*) ایجاد می شود. نمودار*scatter* با استفاده از روش پراکندگی شی *Axes (ax)* ایجاد می شود.

```
X[y == j, feature_pairs[i][0]], X[y == j, feature_pairs[i][1]],
label=f'Class {j}', c=colors[j]
```

در این خط نقاط داده را از فضای ویژگی (*X*) که به کلاس *j* برای جفت *Feature* هایی که انتخاب شده اند اختصاص داده میشود.(یعنی به عنوان مثال :

X[y == j, feature_pairs[i][0]], X[y == j, feature_pairs[i][1]] در واقع در اینجا *X* هایی که *label* آنها برابر *y==j* و جزو *Feature* مورد نظر هستند انتخاب میشود.)

پارامتر *label* بر اساس برچسب کلاس (*j*) یک برچسب به نمودار *scatter* اختصاص می دهد.

پارامتر *c* رنگ نمودار *scatter* را بر اساس برچسب کلاس (*j*) با استفاده از رنگ مشخص شده در آرایه رنگ ها تنظیم می کند.

پس از رسم تمام نمودارهای *scatter* برای *feature pairs* های فعلی، *label* ها را برای محور *x* و محور *y* *subplot(ax)* تنظیم می کند.

label ها به صورت پویا بر اساس *feature index* در *feature_pairs* تولید می شوند.

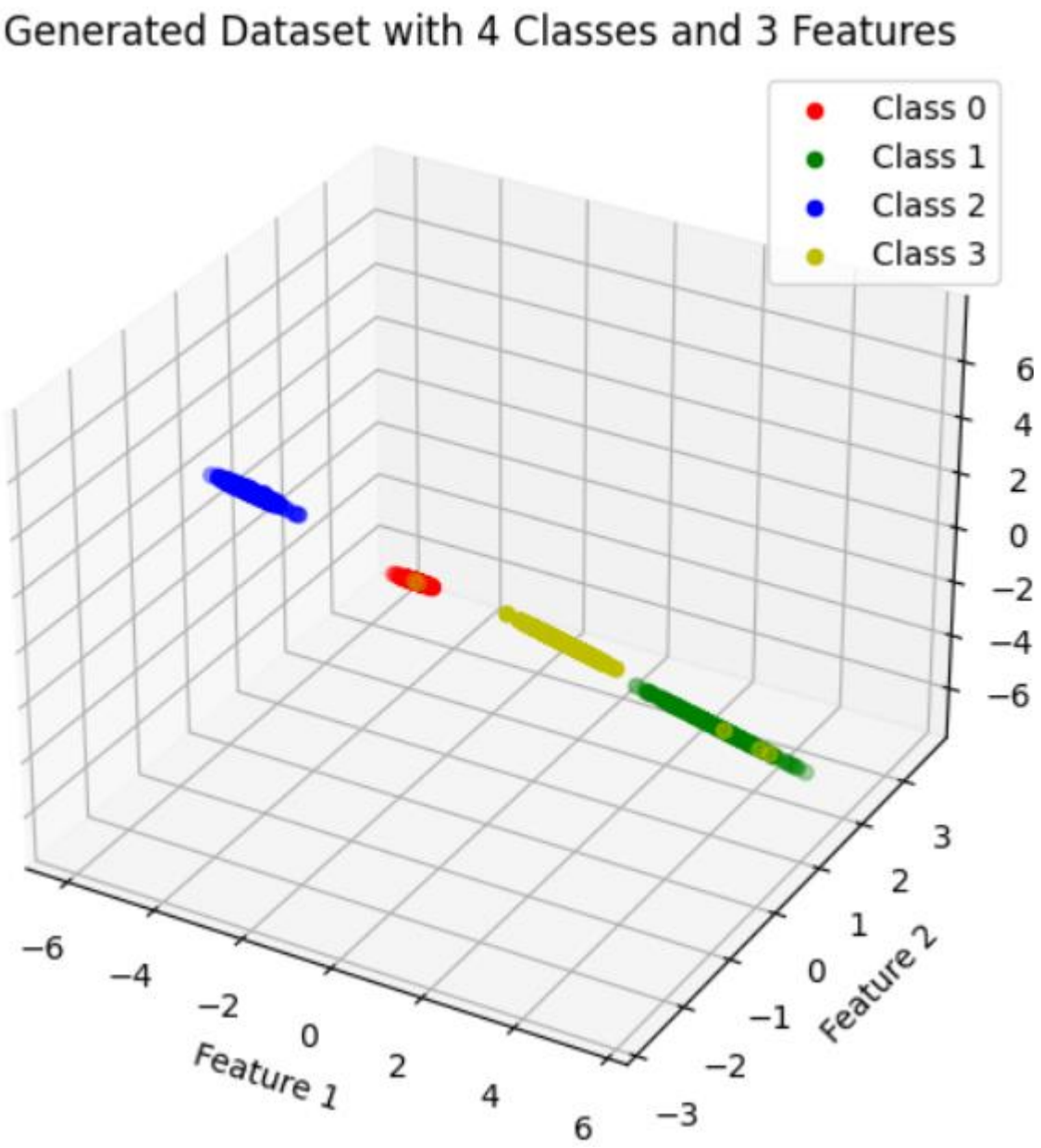
متد *legend* برای نمایش *legend* نمودار فراخوانی می شود که برچسب های کلاس مربوط به رنگ های استفاده شده در *scatter* را نشان می دهد.

تابع *suptitle* در *matplotlib* برای افزودن یک عنوان در بالای کل شکل استفاده می‌شود.

تابع *tight_layout* در *matplotlib* موقعیت *subplots* را در شکل تنظیم می کند تا از همپوشانی عناصر جلوگیری کند و اطمینان حاصل کند که همه عناصر به درستی قابل مشاهده هستند.

و در آخر هم *plt.show()* شکل(*figure*) که همه *subplot* ها را داراست نشان میدهد.

یک نمودار دیگر نیز برای نشان دادن داده ها به صورت ۳بعدی داریم که بدین شکل است :



توضیح کد(قسمت هایی که مانند کد قبلی نیست) :

```
rom sklearn.datasets import make_classification
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

X, y = make_classification(n_samples=1000, n_redundant=1,
n_clusters_per_class=1, class_sep=4, n_features=3, n_classes=4,
random_state=64)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
```

```
colors = ['r', 'g', 'b', 'y']
for i, color in zip(range(4), colors):
    ax.scatter(X[y == i, 0], X[y == i, 1], X[y == i, 2], c=color,
label=f'Class {i}')

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_zlabel('Feature 3')
ax.legend()
plt.title('Generated Dataset with 4 Classes and 3 Features')
plt.show()
```

mpl_toolkits.plot3d: این ماژول در کتابخانه *Matplotlib* است که ابزارهایی برای ایجاد نمودارهای سه بعدی ارائه می دهد.

Axes3D: این یک کلاس در ماژول *mpl_toolkits.plot3d* است. این یک شی محور سه بعدی را در شکل *Matplotlib* نشان می دهد.

plt.figure(figsize=(8, 6)) یک شی شکل جدید برای رسم ایجاد می کند.

پارامتر *figsize* عرض و ارتفاع شکل را بر حسب اینچ مشخص می کند. در این حالت عرض 8 اینچ و ارتفاع 6 اینچ تعیین می شود.

fig.add_subplot(111, projection='3d') یک *subplot* به شکل اضافه می کند.

پارامترهای 111 نشان می‌دهند که شبکه پلات فرعی دارای 1 ردیف، 1 ستون است و این اولین *subplot* است.

پارامتر *'projection='3d* مشخص می کند که این *subplot* یک نمودار سه بعدی خواهد بود.

متغیر *ax* به این *subplot* اختصاص داده شده است که به ما امکان می دهد تغییرات بیشتری را انجام دهیم و داده های ترسیمی را روی آن ترسیم کنیم.

این حلقه بر روی هر کلاس (*i*) و رنگ متناظر آن با استفاده از تابع *zip* تکرار می‌شود، که عناصر را از چندین تکرار جفت می‌کند.(*zip* یک تابع در پایتون است که لیست تاپل یا رشته به عنوان ورودی میگیرد.)

۳)

ما در اینجا از ۲ طبقه بند خطی استفاده کردیم :۱) *Logistic regression* ۲) *SGD classifier*

ما در ۴ حالت داده مدلسازی را انجام دادیم :

حالت اول : *N_redundant = 1 & Random state = 64*

```
Logistic Regression:
Best Parameters: {'C': 0.001, 'max_iter': 100}
Training Accuracy: 0.9925
Testing Accuracy: 0.975

SGD Classifier:
Best Parameters: {'alpha': 0.0001, 'eta0': 0.01, 'max_iter': 100}
Training Accuracy: 0.9925
Testing Accuracy: 0.975
```

حالت دوم :*N_redundant = 0 & Random state = 64*

```
Logistic Regression:
Best Parameters: {'C': 0.001, 'max_iter': 100}
Training Accuracy: 0.99375
Testing Accuracy: 0.98

SGD Classifier:
Best Parameters: {'alpha': 0.0001, 'eta0': 0.01, 'max_iter': 100}
Training Accuracy: 0.99375
Testing Accuracy: 0.98
```

حالت سوم : *N_redundant = 1 & Random state = 32*

```
Logistic Regression:
Best Parameters: {'C': 0.001, 'max_iter': 100}
Training Accuracy: 0.9975
Testing Accuracy: 0.995

SGD Classifier:
Best Parameters: {'alpha': 0.0001, 'eta0': 0.01, 'max_iter': 100}
Training Accuracy: 0.9975
Testing Accuracy: 0.995
```

حالت چهارم : *N_redundant = 0 & Random state = 32*

```
Logistic Regression:
Best Parameters: {'C': 0.001, 'max_iter': 100}
Training Accuracy: 0.98875
Testing Accuracy: 0.995

SGD Classifier:
Best Parameters: {'alpha': 0.0001, 'eta0': 0.01, 'max_iter': 100}
Training Accuracy: 0.98875
Testing Accuracy: 0.995
```

در این بخش همینطور که مشاهده میکنید بهترین نتیجه مربوط به حالت سوم و چهارم میباشد ولی دربین حالت های اول و دوم که داده های آنرا در بخش قبل ***plot***کردیم بهترین حالت حالت دوم میباشد.(دلیل این اتفاق میتواند اینطور باشد که داده های کمی در اختیار داریم وگر نه بصورت عادی چون بین داده های حالت دوم در فضای حالت ***overlap*** موجود است باید حالت دوم نتایج بهتری را به نمایش بگذارد.)

در حالت سوم و چهارم نیز نتیجه در حالت ارزیابی برابرند که میتواند نشان دهد که یکی از دلایل این موضوع از تعداد داده های کم میباشد زیرا مدل ما توانایی بدست آوردن ***pattern*** مشخص را در داده کم ندارد وگر نه باد حالتی که داده های کلاس ها ***overlap*** نداشته باشند نتیجه بهتری داشته باشد.

در این قسمت بدین شکل فراپارامتر ها را معین کردیم که دسته ای از آنها را آزمایش کرده و بهترین را انتخاب کردیم :

```
# Define classifiers
classifiers = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'SGD Classifier': SGDClassifier(max_iter=1000)
}

# Define parameter grids for hyperparameter tuning
param_grids = {
    'Logistic Regression': {'C': [0.001, 0.01, 0.1, 1, 10] , 'max_iter': [100, 500, 1000]},
    'SGD Classifier': {'alpha': [0.0001, 0.001, 0.01, 0.1], 'eta0': [0.01, 0.1, 1], 'max_iter': [100, 500, 1000]}
}
```

همینجور که در کد بالا مشاهده میکنید اولین کاری که کردیم مشخص کردن طبقه بندی کننده ها بود.در عمل بعدی مجموعه ای از پارامتر های آنهارا معلوم کردیم مانند ماکسیمم تکرار و ***regularization parameter(c)*** برای ***Logistic regression***. بعد از آن برای ***SGD*** پارامتر های ***alpha*** و ***eta*** و ماکسیمم تکرار را معلوم کردیم.

همانطور که در نتایج مشاهده کردید نتیجه بهترین پارامتر بعد آموزش مشاهده شده است.

تکنیک هایی که برای بهبود نتیجه استفاده کردیم اولی ***standardization*** داده ها توسط دستور ***standardscalar*** میباشد در این کار ما داده ها را در مقیاس کوچک در میانگین ۰ و واریانس ۱ در آوردیم.(***preprocessing***)

عمل بعدی هم معلوم کردن مجموعه ای از پارامتر ها و پیدا کردن بهترین پارامتر توسط کتابخانه***GridsearchCv*** میباشد.(***Hyperparameter Tuning***) ***GridSearchCV*** تکنیکی برای تنظیم هایپرپارامتر است که به طور جامع در یک شبکه پارامتر مشخص شده جستجو می کند و عملکرد مدل را برای هر ترکیبی از فراپارامترها ارزیابی می کند.

همچنین در روند آموزش برای هر طبقه بندی کننده از ***5 corss-validation*** استفاده کرده که شامل تقسیم داده های آموزشی به پنج تا با اندازه مساوی، استفاده از چهار تا برای آموزش و یک برابر برای اعتبار سنجی(***validation***) در هر تکرار است. جستجوی شبکه‌ای در هر فولد انجام می‌شود تا بهترین هایپرپارامترها را بر اساس عملکرد مجموعه اعتبارسنجی پیدا کند.

کد آن بدین شکل است :

```
# Train and evaluate classifiers
results = {}
for name, clf in classifiers.items():
    print(f"Training {name}...")
    grid_search = GridSearchCV(clf, param_grids[name], cv=5, n_jobs=-1)
    grid_search.fit(X_train, y_train)
    best_clf = grid_search.best_estimator_

    # Training phase accuracy
    y_train_pred = best_clf.predict(X_train)
    train_accuracy = accuracy_score(y_train, y_train_pred)

    # Testing phase accuracy
    y_test_pred = best_clf.predict(X_test)
    test_accuracy = accuracy_score(y_test, y_test_pred)

    results[name] = {'Best Parameters': grid_search.best_params_,
                     'Training Accuracy': train_accuracy,
                     'Testing Accuracy': test_accuracy}
```

اخط یک *dictionary* خالی به نام *results* را راه اندازی می کند که نتایج ارزیابی را برای هر طبقه بندی کننده ذخیره می کند.

خط بعدی حلقه روی هر طبقه‌بندی کننده (*clf*) در طبقه‌بندی‌کننده تکرار می‌شود، جایی که هر طبقه‌بندی کننده با یک *name* مرتبط است.

خط بعدی پیامی را چاپ می کند که نشان می دهد آموزش در شرف شروع برای طبقه بندی کننده فعلی است.

در داخل حلقه، یک شی *GridSearchCV* برای تنظیم هایپرپارامتر ایجاد شده است. برای استفاده از تمام هسته‌های *CPU* موجود، طبقه‌بندی‌کننده (*clf*)، شبکه پارامتر مربوطه (*param_grids[name]*)، تعداد *cross-validation* (*cv=5*) و *n_jobs=-1* نیاز است(در واقع این قسمت برای استفاده کامل از *CPU* میباشد.). سپس متد *fit()* از *grid search object* را با داده های آموزشی (*X_train* و *y_train*) منطبق می کند.

پس از برازش *grid seach object*، بهترین طبقه‌بندی کننده با فراپارامترهای بهینه با استفاده از ویژگی *best_estimator_* به دست می‌آید.

پیش‌بینی‌های طبقه‌بندی‌کننده با استفاده از بهترین تخمین‌گر (*best_clf*) روی داده‌های آموزشی (*X_train*) تولید می‌شوند. سپس دقت پیش‌بینی‌های مدل در مجموعه آموزشی با استفاده از تابع *accuracy_score* محاسبه می‌شود.

به همین ترتیب، پیش‌بینی‌ها بر روی داده‌های آزمون (*X_test*) با استفاده از بهترین تخمین‌گر انجام می‌شود و دقت پیش‌بینی‌های مدل روی مجموعه آزمون محاسبه می‌شود.

نتایج طبقه‌بندی‌کننده فعلی (*name*) در *results* ذخیره می‌شوند. این شامل بهترین هایپرپارامترهای یافت شده در طول جستجوی شبکه، و همچنین دقت آموزش و آزمایش است.

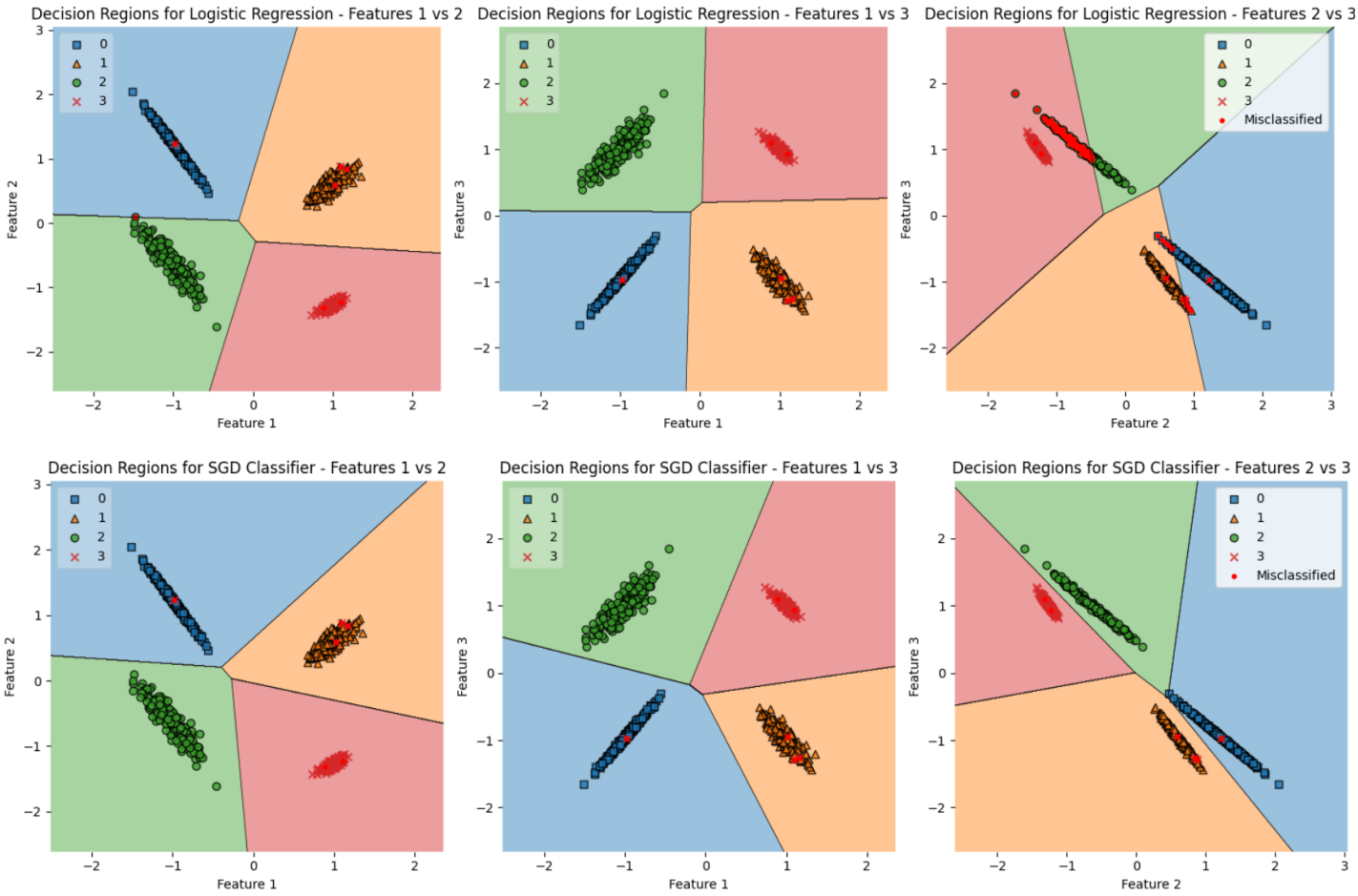
در نهایت، نتایج ارزیابی برای هر طبقه‌بندی نمایش داده می‌شود. برای هر طبقه‌بندی، بهترین پارامترها، دقت آموزش و دقت تست چاپ می‌شوند. حلقه روی هر جفت در *results* تکرار می شود، جایی که *name* نشان دهنده نام طبقه بندی کننده است، و *result* نشان دهنده *results[name]* است.

۴)

ما در اینجا مدل بدست آمده از حالت اول بخش قبل را استفاده میکنیم.کد مانند قسمت های قبل میباشد با این تفاوت که مدل مان را با آموزش بر روی داده های *Feature* های انتخابی و *plot* بر روی آن بدست آوردیم :

```
plt.figure(figsize=(15, 5))
for i, feature_pair in enumerate([(0, 1), (0, 2), (1, 2)]):
    # Train logistic regression only on selected features
    X_train_pair = X_train_scaled[:, feature_pair]
    best_clf.fit(X_train_pair, y_train)
```

و سپس با دستور *plot_desicsion_regions* داده ها را از هم جدا کردیم :



همینطور که میبینید *SGD* بهتر عمل کرده است(برای *partition* بندی)

ضرب در هایی که مثلا در شکل ۲ در کلاس ۰ هستند در واقع داده هایی هستند که *misclassified* هستند.برای معلوم کردن این داده ها بدین شکل عمل کردیم :

```
# Highlight misclassified data points
misclassified_mask = y_train != best_clf.predict(X_train_pair)
plt.scatter(X_train_pair[misclassified_mask, 0], X_train_pair[misclassified_mask, 1],
            marker='o', s=10, c='red', label='Misclassified')
```

y_train != best_clf.predict(X_train_pair) برچسب های واقعی (*y_train*) را با برچسب های پیش بینی شده مقایسه می کند (*best_clf.predict(X_train_pair)*).

این مقایسه منجر به یک آرایه بولی (*misclassified_mask*) می‌شود که در آن *True* نشان می‌دهد که یک نقطه داده اشتباه طبقه‌بندی شده است (برچسب پیش‌بینی‌شده با برچسب واقعی مطابقت ندارد)، و *False* نشان‌دهنده نقاط داده به‌درستی طبقه‌بندی شده است.

X_train_pair[misclassified_mask, 0] و *X_train_pair[misclassified_mask, 1] ,* *Feature* های نقاط داده طبقه بندی شده اشتباه را استخراج می کنند.

Marker='o' سبک نشانگر را برای نقاط طبقه بندی شده روی *'o'* تنظیم می کند که در واقع به صورت دایره است.

s=10 اندازه نشانگر را مشخص می کند.

c='red' رنگ نقاط طبقه بندی شده اشتباه را قرمز می کند.

label='Misclassified' نقاط طبقه بندی شده اشتباه را برای *legend* برچسب گذاری می کند.

۵)

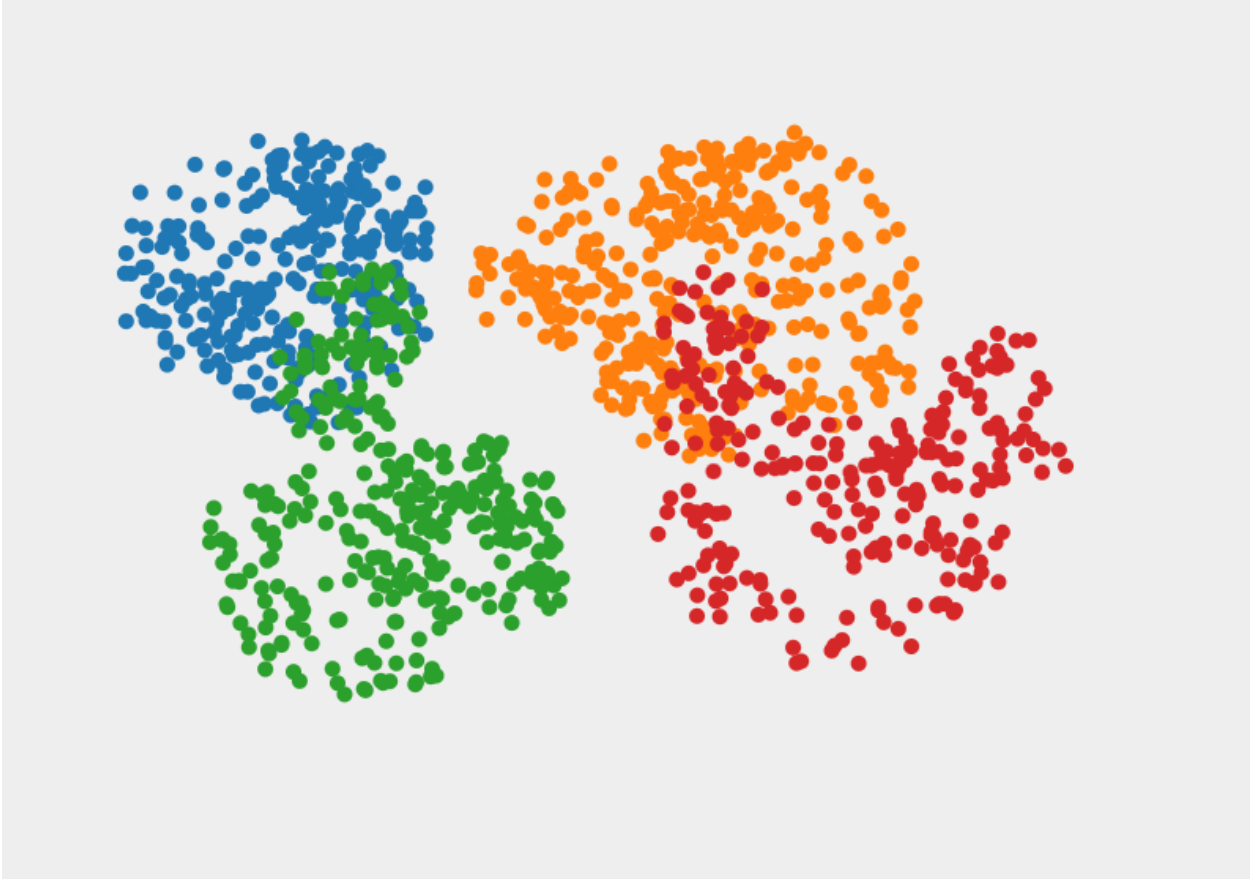
خوب ما داده ها را بر اساس لینکی که قرار داده شد کشیده و در ۴ کلاس با *2 Feature* بدست آورده ایم و آنها را به شکل *dataframe* ذخیره کرده ایم(کتابخانه *pandas* را وارد کرده ایم).کد پ نتایج بدین صورت میباشد:

اول با این دستور *drawdata* را نصب کرده ایم :

```
!pip install drawdata
```

سپس *pandas* و*drawdata* را وارد کرده و برنامه را راه اندازی میکنیم :

```
from drawdata import ScatterWidget
import pandas as pd
widget = ScatterWidget()
widget
```



بعد از آن دیتا ها را به صورت دیتافریم بدست آورده و ابعاد آن را نیز بدست آورده ایم سپس داده های ***feature*** ها مورد نظر را جدا کرده و بعد از آن داده های ستون ***label*** را به عنوان ***target*** جدا میکنیم و در آخر داده ها را به صورت ***csv*** ذخیره کرده تا بتوان بعدا از آن استفاده کرد :

```
# Get the drawn data as a dataframe
df=widget.data_as_pandas
print(df.shape)
#widget.data_as_polars
X = df[['x' , 'y']]
y = df['label']

df.to_csv('/content/drawdata.csv', index=False)
```

```
(1248, 4)
```

تغییراتی را در مدل های خود انجام میدهیم که بدین شکل است :

```
    'SGD Classifier': SGDClassifier(max_iter=5000)
    'SGD Classifier': {'alpha': [0.0001, 0.001, 0.01, 0.1], 'eta0': [0.01,
0.1, 1 , 10], 'max_iter': [100, 500, 1000 , 5000]}
```

سپس مانند قبل مدل های خود را اجرا میکنیم که نتیجه بدین شکل است :

```
Logistic Regression:
Best Parameters: {'C': 10, 'max_iter': 100}
Training Accuracy: 0.8797595190380761
Testing Accuracy: 0.876

SGD Classifier:
Best Parameters: {'alpha': 0.001, 'eta0': 0.01, 'max_iter': 500}
Training Accuracy: 0.8396793587174348
Testing Accuracy: 0.836
```

همینطور که مشاهده میکنید در هر ۲ حالت قسمت *training accuracy* بیشتر از *testing accuracy* میباشد که نشان دهنده سالم تر بودن داده ها نسبت به قسمت ۲ سوال میباشد.

در *Logistic Regression* همینطور که میبینید *C* یا همان *regularization parameter* و*alpha* بیشتراز حالت های قبل را دارا میباشد(*C* بیشترین حالت را دارد) این نشان دهنده این است که به مدل اجازه می‌دهد تا داده‌های آموزشی را با دقت بیشتری مطابقت دهد. این می تواند منجر به واریانس بالاتر و *overfitting* شود.

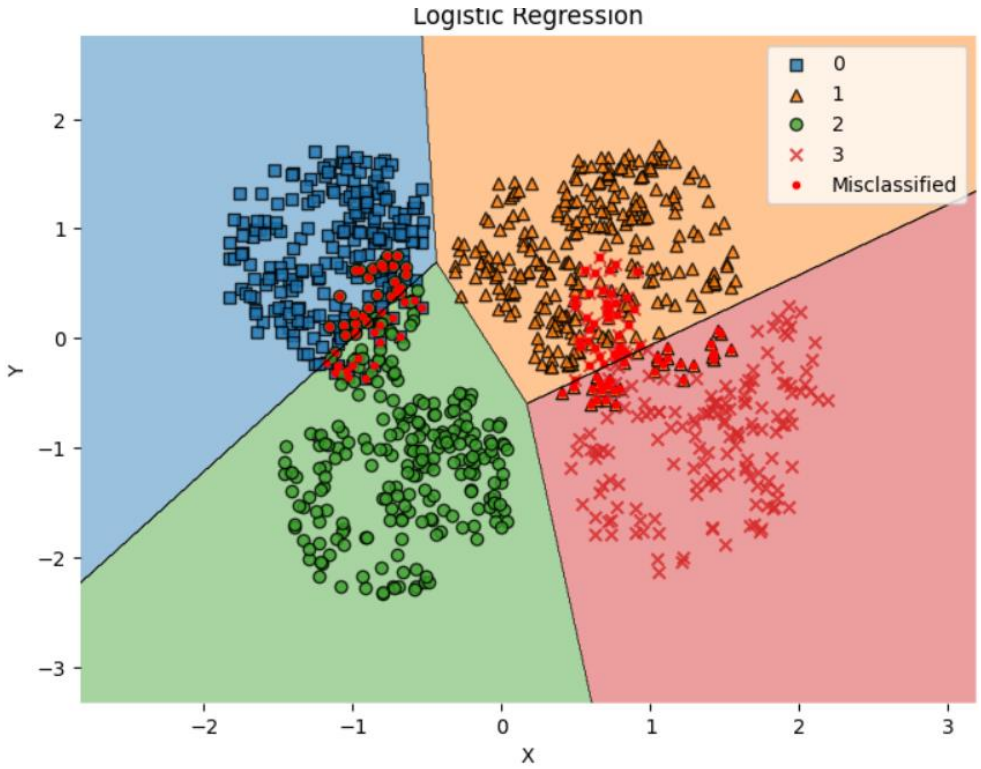
اگر در *SGD , eta*(*learning rate*) بیشتر از قبل (بخش ۲) میشد که این میتواند نشان دهنده به به‌روزرسانی‌های بزرگ‌تر برای پارامترهای مدل می‌شود که می‌تواند روند یادگیری را تسریع کند. با این حال، ممکن است به بی‌ثباتی نیز منجر شود، به‌ویژه اگر نرخ یادگیری بیش از حد بالا باشد، که باعث می‌شود مدل از راه‌حل بهینه فراتر رود(*Overshoot*).

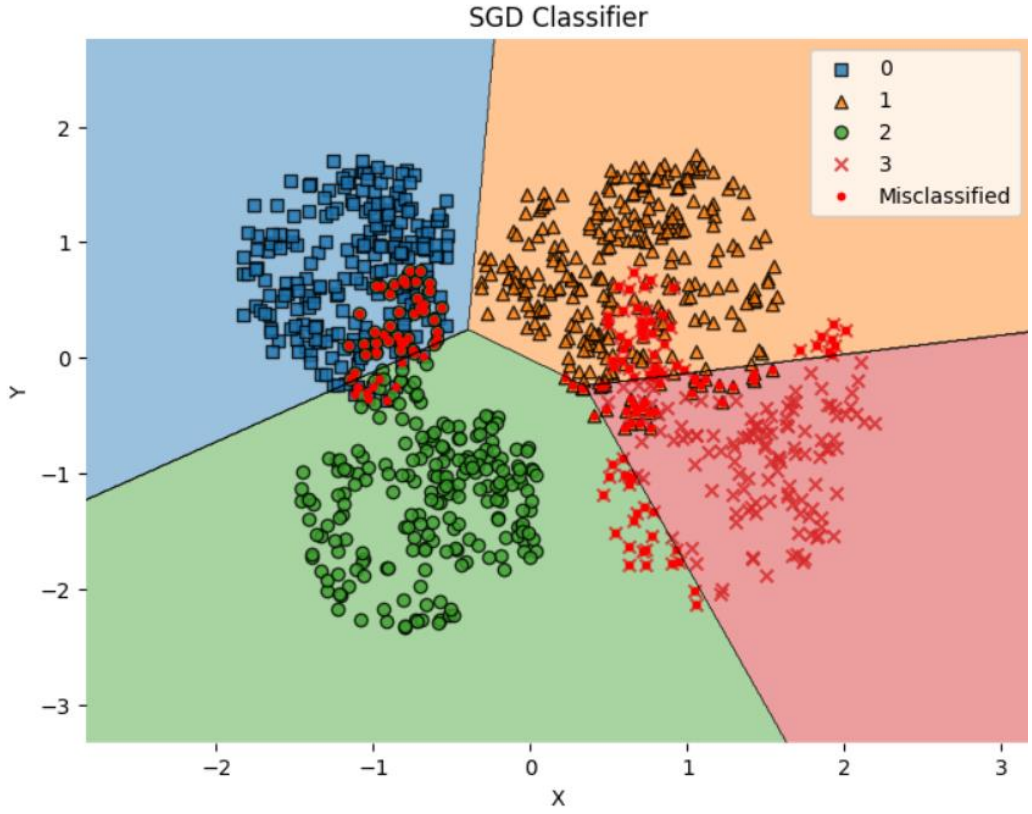
تنها فرق کد این قسمت با قسمت های قبلی اینست که کلاس هایمان به صورت لیبل بندی شده میباشد(مانند *a* ,*b* ,...) ما از کتابخانه *sklearn.preprocessing , LabelEncoder* را وارد میکنیم و *y* های خود را به عدد تبدیل میکنیم(تا بتوانیم آنهارا برای بدست آوردن *decision boundary* رسم کنیم):

```
# Initialize the LabelEncoder
label_encoder = LabelEncoder()
```

```
# Encode the class labels
y_encoded = label_encoder.fit_transform(y)


# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded,
test_size=0.2, random_state=64)
```

برای نشان دادن مرز بین کلاس ها در فضای ***feature*** نیز مانند قبل عمل میکنیم :




SGD Classifier
0
1
2
3
Misclassified
2
1
0
−1
−2
−3
Y
−2
−1
0
1
2
3
X

سوال ۲)

۱)

مجموعه داده بلبرینگ ***CWRU*** یک مجموعه داده شناخته شده در زمینه تشخیص عیب است، به ویژه در نظارت بر سلامت ماشین آلات و تعمیر و نگهداری مبتنی بر شرایط. این مجموعه داده توسط مرکز داده‌های باربری دانشگاه ***Case Western Reserve (CWRU)*** برای اهداف تحقیقاتی در زمینه یادگیری ماشین و نگهداری پیش‌بینی‌کننده جمع‌آوری شده است.

در اینجا یک نمای کلی از مجموعه داده است:

1. ***Goal***: هدف اصلی مجموعه داده های باربری ***CWRU*** تسهیل تحقیق و توسعه در تشخیص عیب و الگوریتم های نگهداری پیش بینی شده است. با ارائه داده‌های ارتعاش در دنیای واقعی از یاتاقان‌های معیوب و سالم، محققان می‌توانند الگوریتم‌هایی را برای شناسایی و طبقه‌بندی انواع مختلف خطاها در یاتاقان‌ها، مانند ***inner race***، ***outer race*** و گسل توپ، توسعه دهند و آزمایش کنند.

2. ***Features*** : مجموعه داده شامل سیگنال های ارتعاشی جمع آوری شده از شتاب سنج های نصب شده بر روی یاتاقان ها در شرایط عملیاتی مختلف است. سیگنال‌های ارتعاشی با فرکانس بالا نمونه‌برداری می‌شوند تا اطلاعات دقیقی در مورد وضعیت بلبرینگ دریافت کنند. علاوه بر این، مجموعه داده شامل ابرداده هایی(***metadata***) مانند نوع خطا، شدت خطا، سرعت چرخش یاتاقان و بار اعمال شده به یاتاقان است.

3. ***Modes*** : مجموعه داده‌های باربری ***CWRU*** حالت‌ها یا زیرمجموعه‌های متفاوتی از داده‌ها را ارائه می‌دهد که هر کدام یک تنظیم یا شرایط آزمایشی خاص را نشان می‌دهند. این حالت ها عبارتند از:

- اطلاعات خطای یاتاقان انتهایی درایو(***Drive-End Bearing Fault Data***): سیگنال های ارتعاشی جمع آوری شده از شتاب سنج های نصب شده بر روی یاتاقان انتهایی موتور القایی با شدت خطاهای مختلف (به عنوان مثال، 0.007 اینچ، 0.014 اینچ).

- اطلاعات خطای یاتاقان انتهای فن(***Fan-End Bearing Fault Data***): سیگنال های ارتعاشی جمع آوری شده از شتاب سنج های نصب شده بر روی یاتاقان انتهایی فن یک موتور القایی با شدت خطاهای مختلف.

- داده‌های پایه عادی(***Normal Baseline Data***) : سیگنال‌های ارتعاشی که از یاتاقان‌های سالم در شرایط عملیاتی عادی جمع‌آوری می‌شوند، به عنوان خط پایه برای مقایسه با داده‌های معیوب عمل می‌کنند.

- داده بار افقی(*Horizontal Load Data*) : سیگنال های ارتعاشی جمع آوری شده در حین اعمال بار افقی بر یاتاقان ها، شبیه سازی شرایط عملیاتی مختلف.

- داده های بار عمودی(*Vertical Load Data*): سیگنال های ارتعاشی جمع آوری شده در حین اعمال بار عمودی بر یاتاقان ها، داده های اضافی را برای الگوریتم های تشخیص عیب ارائه می دهد.

- داده های مختلف قطر خطا(*Different Fault Diameter Data*): سیگنال های ارتعاشی جمع آوری شده از یاتاقان های دارای خطاهایی با قطرهای مختلف، به محققان اجازه می دهد تا تأثیر اندازه خطا را بر دقت تشخیص مطالعه کنند.

با ارائه این حالت‌های مختلف، مجموعه داده محققین را قادر می‌سازد تا عملکرد الگوریتم‌های تشخیص عیب را تحت شرایط و سناریوهای مختلف ارزیابی کنند و در نهایت پیشرفت‌های پیشرفته را در نگهداری پیش‌بینی‌کننده و نظارت بر سلامت ماشین‌آلات پیش ببرند.

داده ها را از سایت مورد نظر دانلود میکنیم.داده ها شامل ۲ دسته داده های نرمال و دارای ***fault*** میباشند.داده های سالم دارای ۲ قسمت ***DE*** و ***FE*** میباشند.داده های خطا دار دارای ***DE,FE,BA*** میباشند.ما در اینجا داده های ***DE*** را در ۲ کلاس بررسی میکنیم.

۲) آ)

اول از همه داده ها را با دستور ***gdown*** دانلود میکنیم.سپس مقدار آنها را ذخیره میکنیم. مثلا :

```
X097_DE_time = pd.read_csv('X097_DE_time.csv',header=None).values
```

خوب تسکی که از ما خواسته شده اینگونه میباشد که باید ***M*** نمونه که در اینجا مقدار ۱۰۰ را در نظر گرفتیم با طول ***N*** (که در اینجا ۲۰۰ میباشد) را از داده های خود جدا کرده (هم داده های سالم و هم معیوب= ***selected_samples***) و این ها را در ماتریس جدید قرار دهیم.(لیبل ها را نیز باید جدا کنیم یعنی داده های سالم لیبل ۰ دارند و نا سالم لیبل ۱)

در واقع در پایان ۲۰۰ لیبل و یک ماتریس ۲۰۰*۲۰۰ خواهیم داشت.

تنها کد جدید بدین شکل میباشد :

```
    selected_samples[i] = np.squeeze( X097_DE_time[start_point:start_point
+ N])
```

که در واقع در اینجا داده ها را که به صورت آرایه هستند جدا میکند و در ***selected_samples*** میریزد.(مثلا ***shape*** داده ها را از ۱*۲۰۰ به ۲۰۰ تبدیل میکند.)

که نتیجه بدین شکل میباشد :

```
print(selected_samples.shape)
print(labels.shape)
```

```
(200, 200)
(200,)
```

ب)

استخراج ویژگی نقش مهمی در یادگیری ماشین دارد، به‌ویژه در سناریوهایی که داده‌های خام حاوی تعداد زیادی متغیر یا ویژگی است. این فرآیند شامل تبدیل داده های خام به مجموعه ای کاهش یافته از ویژگی های مرتبط است که می تواند به طور موثر الگوها یا ویژگی های اساسی داده ها را نشان دهد. در اینجا توضیحی در مورد اهمیت استخراج ویژگی ارائه شده است:

کاهش ابعاد: استخراج ویژگی با انتخاب یا ایجاد زیرمجموعه‌ای از ویژگی‌هایی که مهم‌ترین اطلاعات را در بر می‌گیرد، به کاهش ابعاد داده‌ها کمک می‌کند. این هنگام برخورد با داده‌های با ابعاد بالا حیاتی است، زیرا می‌تواند به بهبود عملکرد مدل، کاهش پیچیدگی محاسباتی و اجتناب از ***curse of dimensionality*** منجر شود.

عملکرد مدل بهبود یافته: با تمرکز بر ویژگی‌های مرتبط و حذف نویز یا اطلاعات نامربوط، استخراج ویژگی می‌تواند به تعمیم بهتر و بهبود عملکرد مدل‌های یادگیری ماشین منجر شود. این به افزایش تفسیرپذیری مدل، کاهش بیش از حد برازش، و قوی‌تر کردن مدل‌ها برای داده‌های دیده نشده کمک می‌کند.

محاسبه کارآمد: استخراج ویژگی های اطلاعاتی منابع محاسباتی مورد نیاز برای آموزش و استنتاج را کاهش می دهد. با ویژگی‌های کمتر، می‌توان مدل‌ها را سریع‌تر آموزش داد و آن‌ها را برای کاربردهای بلادرنگ یا در مقیاس بزرگ مناسب‌تر می‌کند.

تفسیرپذیری پیشرفته: استخراج ویژگی می‌تواند الگوها یا روابط معنی‌داری را در داده‌ها آشکار کند و تفسیر رفتار مدل و درک عوامل اساسی پیش‌بینی‌ها را برای انسان آسان‌تر کند.

کاهش نویز: داده های خام اغلب حاوی نویز یا اطلاعات نامربوطی هستند که می تواند عملکرد مدل را مختل کند. هدف تکنیک های استخراج ویژگی شناسایی و حذف چنین نویزهایی است که منجر به پیش بینی های دقیق تر و قابل اعتمادتر می شود.

Handling Redundancy: استخراج ویژگی به شناسایی و حذف ویژگی‌های اضافی یا بسیار همبسته کمک می‌کند، که می‌تواند باعث ایجاد مشکلات چند خطی و کاهش عملکرد مدل‌هایی مانند رگرسیون خطی شود.

Domain-Specific Knowledge Incopration : استخراج ویژگی اجازه می دهد تا دانش خاص دامنه در فرآیند مدل سازی گنجانده شود. متخصصان دامنه می توانند ویژگی های مرتبطی را که برای حل مشکلات خاص حیاتی هستند شناسایی کنند و به راه حل های یادگیری ماشینی موثرتر منجر شوند.

تجسم داده ها: ویژگی های استخراج شده را می توان برای به دست آوردن بینش در مورد ساختار زیربنایی داده ها تجسم کرد. تکنیک هایی مانند کاهش ابعاد (به عنوان مثال، *PCA*) می تواند به تجسم داده های با ابعاد بالا در فضاهای با ابعاد پایین تر، تسهیل درک بهتر و تجزیه و تحلیل داده های اکتشافی کمک کند.

خوب در این قسمت ما با *9 Features* کارمان را انجام میدهیم که بدین شکل میباشند :

```python
# Feature extraction methods
def shape_factor(data):
    return np.sqrt(np.mean(np.square(data))) / np.mean(np.abs(data))

def impact_factor(data):
    return np.max(data)/ np.mean(np.abs(data))

def Crest_Factor(data):
    return  np.max(data) / np.sqrt(np.mean(np.square(data)))

def absolute_mean(data):
    return np.mean(np.abs(data))

def root_mean_square(data):
    return np.sqrt(np.mean(np.square(data)))

def impulse_factor(data):
    return np.max(np.abs(data)) / np.mean(np.abs(data))

def standard_deviation(data):
    return np.std(data)

def impact_factor(data):
    return np.max(np.abs(data)) / np.mean(np.abs(data))
```

در اینجا ۸ تای آنها نشان داده شده اند زیرا ۹ امین آنها میانگین است که با دستور ساده ***np.mean(data)*** به سادگی بدست می آید :

```
# Initialize empty array to store extracted features
num_features = 9
num_samples, sample_length = selected_samples.shape
extracted_features = np.zeros((num_samples, num_features))

# Extract features for each sample
for i in range(num_samples):
    sample = selected_samples[i]
    extracted_features[i, 0] = shape_factor(sample)
    extracted_features[i, 1] = impact_factor(sample)
    extracted_features[i, 2] = Crest_Factor(sample)
    extracted_features[i, 3] = np.mean(sample)
    extracted_features[i, 4] = absolute_mean(sample)
    extracted_features[i, 5] = root_mean_square(sample)
    extracted_features[i, 6] = impulse_factor(sample)
    extracted_features[i, 7] = standard_deviation(sample)
    extracted_features[i, 8] = impact_factor(sample)

# Now we have a new dataset with extracted features
extracted_features.shape
```

```
(200, 9)
```

همینطور که میبینید داده ها *(200,9)* تبدیل شده اند.

ج)

مخلوط کردن داده ها یک مرحله پیش پردازش ضروری در یادگیری ماشین است، به ویژه در سناریوهایی که داده ها ممکن است دارای نظم یا ساختار ذاتی باشند. در اینجا چند دلیل کلیدی وجود دارد که چرا به هم زدن داده ها مهم است:

جلوگیری از *Bias* در آموزش: درهم آمیختن تضمین می کند که نمونه داده ها به ترتیب تصادفی در طول آموزش به مدل ارائه می شوند. این تصادفی بودن کمک می کند تا مدل از یادگیری هرگونه سوگیری(*bias*) یا الگویی که ممکن است در ترتیب داده ها وجود داشته باشد جلوگیری کند. به عنوان مثال، اگر مجموعه داده بر اساس برچسب‌های کلاس مرتب شده باشد، به هم زدن از یادگیری تکیه بر ترتیب کلاس‌ها توسط مدل جلوگیری می‌کند.

بهبود تعمیم: مخلوط کردن به بهبود توانایی تعمیم مدل کمک می کند. هنگامی که داده ها به هم ریخته می شوند، مدل در طول هر دوره آموزشی در معرض نمونه های مختلفی از بخش های مختلف مجموعه داده قرار می گیرد. این به مدل کمک می‌کند تا یاد بگیرد که داده‌های دیده نشده را بهتر تعمیم دهد، که منجر به بهبود عملکرد در داده‌های آزمایشی می‌شود.

اجتناب از ***overfitting*** : هم زدن می تواند به جلوگیری از بیش از حد تناسب کمک کند. هنگام آموزش با داده های مرتب شده، مدل ممکن است به طور ناخواسته یاد بگیرد که الگوهای مخصوص به ترتیب نمونه های آموزشی را به جای یادگیری الگوهای معنی دار در داده ها به خاطر بسپارد. به هم ریختگی هر گونه همبستگی کاذبی را که ممکن است به دلیل ترتیب داده ها وجود داشته باشد، مختل می کند و منجر به مدلی می شود که بهتر به داده های جدید تعمیم می یابد.

اطمینان از استقلال نمونه ها: مخلوط کردن تضمین می کند که هر نمونه در مجموعه داده مستقل و به طور یکسان توزیع شده است (***i.i.d***). این فرض استقلال در بسیاری از الگوریتم‌های یادگیری ماشین، به ویژه الگوریتم‌هایی که فرض می‌کنند نمونه‌ها از یک توزیع گرفته شده‌اند، اساسی است. مخلوط کردن با حذف هر گونه وابستگی احتمالی که ممکن است به دلیل ترتیب داده ها وجود داشته باشد، به حفظ این استقلال کمک می کند.

بهبود بهینه سازی: مخلوط کردن داده ها می تواند منجر به بهینه سازی پایدارتر در طول آموزش شود. در الگوریتم‌های بهینه‌سازی تصادفی مانند نزول گرادیان تصادفی (***SGD***)، مخلوط کردن به شکستن هر گونه همبستگی بین دسته‌های متوالی داده کمک می‌کند. این همبستگی منجر به مسیرهای بهینه سازی هموارتر و همگرایی سریعتر به یک راه حل خوب می شود.

خوب در اینجا ما تعداد ۸۰ درصد را به آموزش و ۲۰ درصد داده ها را به ارزیابی اختصاص میدهیم بدین صورت که ستون آخر (لیبل ها یعنی همان ۰ و ۱) در واقع داده های ***y*** ما میباشند و بقیه ستون ها داده های ***x*** ما و سپس با درصد گفته شده داده ها را تقسیم بندی میکنیم :

```
# Combine features and labels
data_with_labels = np.column_stack((extracted_features, labels))
print(data_with_labels.shape)
# Shuffle the data
np.random.shuffle(data_with_labels)
data_with_labels
```

```
(200, 10)
array([[1.35169662, 5.89001373, 4.35749682, ..., 0.27870677, 5.89001373,
        1.        ],
       [1.46229815, 6.78567868, 4.64042076, ..., 0.30938286, 6.78567868,
        1.        ],
       [1.2661485 , 2.98021595, 2.35376495, ..., 0.05432436, 2.98021595,
        0.        ],
       ...,
       [1.41566487, 5.78591659, 4.0870666 , ..., 0.25509884, 5.78591659,
        1.        ],
       [1.19317073, 2.44071645, 2.04557184, ..., 0.07809902, 2.44071645,
        0.        ],
```

همینجور که در بالا مشاهده میکنید ما ابتدا با دستور *np.column_stack* لیبل و ماتریس جدید *extracted_features*(که در واقع ماتریسی هستش که بعد از اعمال تولید ویژگی درست شده است) را با هم ادغام کرده سپس سطر های داده ها با دستور *np.random.shuffle* بر میزنیم.همینطور که میبینید ستون آخر داده ها در ابتدا ۱ میباشد که این نشان دهنده این است که به درستی بر زده ایم.

```
# Define the division ratio for training and evaluation sets
division_ratio = 0.8  # 80% for training, 20% for evaluation

# Calculate the split index
split_index = int(division_ratio * num_samples)

# Split the data into training and evaluation sets
training_data = data_with_labels[:split_index]
evaluation_data = data_with_labels[split_index:]

# Separate features and labels for training
X_train = training_data[:, :-1]  # Features for training
y_train = training_data[:, -1]   # Labels for training

# Separate features and labels for evaluation
X_eval = evaluation_data[:, :-1]  # Features for evaluation
y_eval = evaluation_data[:, -1]   # Labels for evaluation
```

کد بالا هم برای تقسیم بندی داده ها به *train* و *test* میباشد.دستور

*int(division_ratio * num_samples)* در واقع عدد *integer* به ما تحویل میدهد که درواقع برابر ۱۶۰ میباشد.اول از همه داده ها را به ۲ دسته *training* و *evaluation* تقسیم بندی کردیم سپس ستون آخر را برای *y* و بقیه ستون ها را برای *x* در *training* و *evaluation* تقسیم بندی کردیم.

د)

عادی سازی داده ها یک مرحله پیش پردازش حیاتی در یادگیری ماشینی است که شامل مقیاس بندی ویژگی ها به یک محدوده مشابه است. این فرآیند به چند دلیل مهم است:

بهبود همگرایی: عادی سازی داده ها کمک می کند تا الگوریتم های بهینه سازی مبتنی بر گرادیان، مانند نزول گرادیان، سریعتر همگرا شوند. وقتی ویژگی‌ها در مقیاس‌های مختلف هستند، فرآیند بهینه‌سازی ممکن است برای رسیدن به حداقل تابع تلفات بیشتر طول بکشد زیرا ممکن است مراحل برداشته شده در فضای پارامتر ناهموار باشد. عادی سازی ویژگی ها تضمین می کند که فرآیند بهینه سازی پایدارتر و کارآمدتر است.

بهبود عملکرد مدل: عادی سازی داده ها می تواند عملکرد مدل های یادگیری ماشین را بهبود بخشد، به ویژه مدل هایی که بر معیارهای مبتنی بر فاصله یا تکنیک های منظم سازی متکی هستند. با آوردن ویژگی ها به مقیاس مشابه، مدل می تواند اهمیت نسبی هر ویژگی را بهتر درک کند و پیش بینی های دقیق تری انجام دهد.

دو روش متداول برای عادی سازی داده ها عبارتند از:

مقیاس حداقل حداکثر (نرمال سازی):

مقیاس حداقل حداکثر، داده ها را به یک محدوده ثابت، معمولاً بین 0 و 1 مقیاس می دهد. شکل توزیع اصلی را حفظ می‌کند اما مقادیر را تغییر می‌دهد و مجدداً مقیاس می‌دهد.

فرمول مقیاس بندی حداقل حداکثر به صورت زیر است:

$$X_{scaled} = (X - X_{\min}) \quad /(X_{\max} - X_{\min})$$

این روش به مقادیر پرت حساس است، زیرا داده ها را بر اساس مقادیر حداقل و حداکثر مقیاس بندی می کند.

استانداردسازی (*Z-score normalization*):

استانداردسازی(*standardization*) داده ها را با میانگین 0 و انحراف معیار 1 مقیاس می کند. داده ها را به توزیع نرمال استاندارد با میانگین 0 و انحراف استاندارد 1 تبدیل می کند.

فرمول استاندارد سازی:

$$X_{std} = \frac{X - \mu}{\sigma}$$

این روش کمتر تحت تأثیر عوامل پرت قرار می گیرد، زیرا بر میانگین و انحراف معیار تکیه دارد.

ما در اینجا روش دوم رو اعمال میکنیم که بدین شکل میباشد :

```
# Calculate mean and standard deviation of each feature
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)

# Perform standardization
X_train_standardized = (X_train - mean) / std
```

خوب در وهله اول میانگین داده ها را گرفتیم(با دستور *np.mean*) سپس ***standard deviation*** داده ها را گرفته(*np.std*) سپس در فرمول قرار میدهیم.

ما معمولاً از اطلاعات بخش "ارزیابی" هنگام نرمال سازی داده ها استفاده نمی کنیم زیرا مجموعه ارزیابی باید به عنوان داده های دیده نشده در نظر گرفته شود. ما مجموعه ارزیابی را با استفاده از میانگین و انحراف استاندارد محاسبه‌شده از مجموعه آموزشی نرمال می‌کنیم تا از سازگاری و شبیه‌سازی سناریوهای دنیای واقعی که داده‌های جدید ممکن است توزیع‌های متفاوتی از داده‌های آموزشی داشته باشند، اطمینان حاصل کنیم.

۳)

در این قسمت ما مدل *logistic regression* را پیاده کردیم.مراحل بدین شکل میباشد :

پارامترهای اولیه: پارامترهای وزن و بایاس را برای مدل رگرسیون لجستیک مقدار میدهیم.

Forward propagation : *logit* ها (ترکیب خطی ویژگی ها و وزن ها) و احتمالات پیش بینی شده را با استفاده از تابع لجستیک محاسبه میکنیم.

تابع ضرر(*Loss*): تابع ضرر (به عنوان مثال، آنتروپی متقاطع باینری*BCE*) را برای اندازه گیری تفاوت بین احتمالات پیش بینی شده و برچسب های واقعی تعریف میکنیم.

Back propagation: گرادیان تابع تلفات را با توجه به پارامترها با استفاده از گرادیان نزول محاسبه میکنیم.

به روز رسانی پارامترها: وزن ها و پارامترهای بایاس را با استفاده از گرادیان های محاسبه شده به روز کرده تا تلفات را به حداقل برسانیم.

ارزیابی: عملکرد مدل را بر روی داده های آزمون با استفاده از معیارهای ارزیابی مانند *accuracy*، *precision*، *recall* یا *F1 score* ارزیابی میکنیم.

```
import matplotlib.pyplot as plt

# data
X = X_train_standardized
y = y_train

# Initialize parameters
np.random.seed(0)
W = np.random.randn(9)
b = 0
```

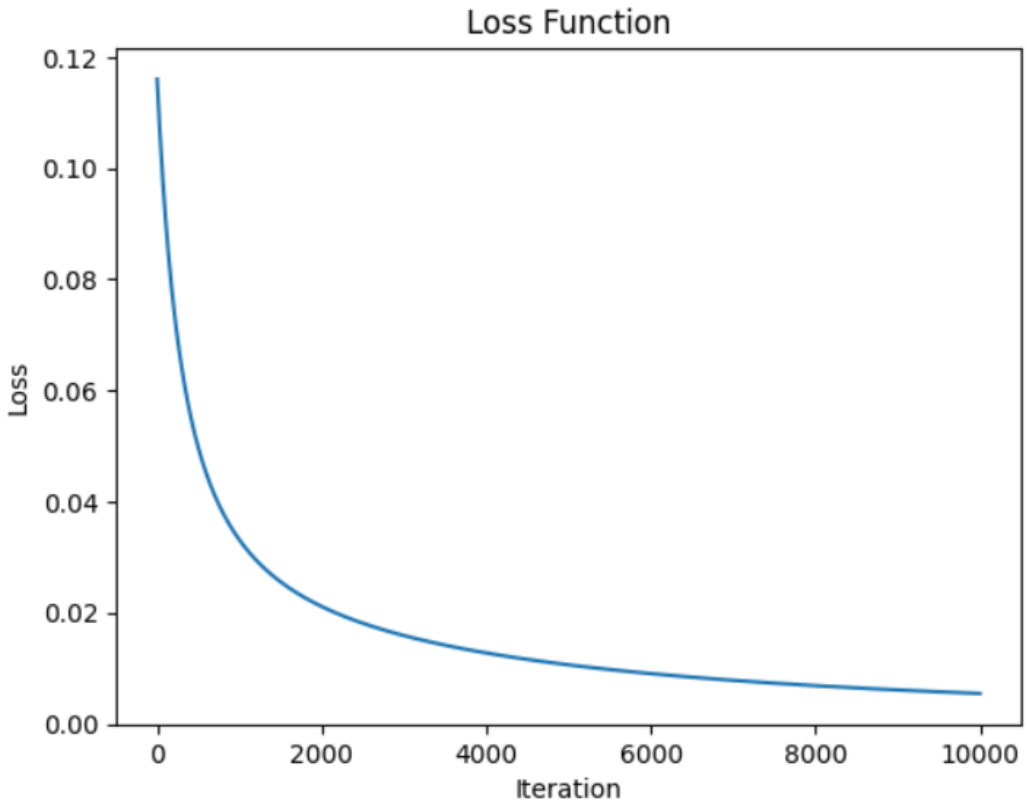
داده ها را تعریف کرده و پارامتر های اولیه را معلوم میکنیم.

```python
# Define logistic function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Forward propagation
def forward_propagation(X, W, b):
    z = np.dot(X, W) + b
    return sigmoid(z)

# Loss function (binary cross-entropy)
def binary_cross_entropy(y_pred, y_true):
    epsilon = 1e-15
    return -np.mean(y_true * np.log(y_pred + epsilon) + (1 - y_true) * np.log(1 - y_pred + epsilon))

# Gradient descent
learning_rate = 0.01
num_iterations = 10000
losses = []
```

تابع های مورد نظر را تعریف میکنیم و پارامتر های گرادیان نزولی را مقدار دهی میکنیم.

```python
for i in range(num_iterations):
    # Forward propagation
    y_pred = forward_propagation(X, W, b)

    # Compute loss
    loss = binary_cross_entropy(y_pred, y)
    losses.append(loss)

    # Backward propagation
    dz = y_pred - y
    dW = np.dot(X.T, dz) / len(X)
    db = np.mean(dz)

    # Update parameters
    W -= learning_rate * dW
    b -= learning_rate * db

# Plot loss function
plt.plot(losses)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Loss Function')
plt.show()
```

سپس در یک حلقه ***for*** که اندازه آن برابر ۱۰۰۰۰ میباشد عملیات های مورد نظر انجام میشود و در آخر تابع ***loss*** نمایش داده میشود که بدین صورت است :



همینجور که میبنید با هر بار تکرار *loss function* مقدار کم تری میدهد که در واقع بدین معنی است که بهتر میتواند خروجی را پیش بینی کند و مدل ما درست عمل کرده است (با داده های *train*) و پارامتر های مورد نظر مانند *w,b* نیز در این حین ذخیره میشوند تا از آن برای داده های *test* استفاده شود.

در حالی که نمودار *loss function* بینش هایی را در مورد فرآیند بهینه سازی ارائه می دهد، برای نتیجه گیری قطعی در مورد عملکرد مدل قبل از مرحله ارزیابی کافی نیست. برای ارزیابی دقیق عملکرد مدل، باید آن را بر روی یک مجموعه داده آزمایشی جداگانه با استفاده از معیارهای ارزیابی مناسب ارزیابی کنیم. این مرحله به بررسی اینکه آیا مدل به خوبی به داده‌های دیده نشده تعمیم می‌یابد کمک می‌کند و بینش قوی‌تری در مورد عملکرد آن ارائه می‌کند.

برای ارزیابی مدل در برابر *testing data* بدین شکل عمل میکنیم :

ما داده های را به ۲ دسته *x_test* و *y_test* تقسیم میکنیم.

ما از *forward propagation* برای به دست آوردن پیش بینی های *y_pred_test* روی داده های آزمون استفاده می کنیم.

ما احتمالات را با استفاده از آستانه 0.5 به پیش بینی های باینری تبدیل می کنیم.

ما *accuracy* را با مقایسه پیش‌بینی‌های باینری با برچسب‌های واقعی محاسبه می‌کنیم.

ما *precision, recall, and F1-score* را با استفاده از پیش‌بینی‌های باینری و برچسب‌های واقعی محاسبه می‌کنیم.

در نهایت، متریک های محاسبه شده را چاپ می کنیم.

نتایج بدین شکل میشود :

```
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1_score)
```

```
Accuracy: 0.475
Precision: 0.475
Recall: 1.0
F1-score: 0.6440677966101694
```

Accuracy: این متریک صحت کلی پیش بینی ها را اندازه گیری می کند. دقت 0.475 به این معنی است که مدل به درستی 47.5 درصد از نمونه ها را در داده های آزمون پیش بینی کرده است.

Precision: این متریک نسبت پیش‌بینی‌های مثبت واقعی را در بین تمام پیش‌بینی‌های مثبت انجام‌شده توسط مدل اندازه‌گیری می‌کند. دقت 0.475 به این معنی است که از همه موارد مثبت پیش بینی شده، تنها 47.5٪ در واقع مثبت بوده اند.

Recall: این متریک نسبت پیش بینی های مثبت واقعی را در بین تمام موارد مثبت واقعی در داده های آزمایش اندازه گیری می کند. 1.0 نشان می دهد که مدل به درستی تمام موارد مثبت واقعی را شناسایی کرده است.

F1 Score: این متریک میانگین هارمونیک *precision* و *recall* است که معیار متعادلی از عملکرد طبقه‌بندی کننده را ارائه می‌دهد. با در نظر گرفتن *precision* و *recall* ، *F1 score* بالاتر، عملکرد کلی بهتری را نشان می دهد. *644*. نسبتاً خوب است که نشان دهنده تعادل منطقی بین *precision* و *recall* است.

بر اساس این معیارها، می‌توان نتیجه گرفت که در حالی که مدل از نظر *recall* (گرفتن همه موارد مثبت در واقع کلاس ۱) خوب عمل می‌کند، اما *precision* نسبتاً کمی دارد (ایجاد نسبت بالایی از پیش‌بینی‌های مثبت کاذب). این نشان می دهد که مدل ممکن است در پیش بینی موارد مثبت بیش از حد حساس باشد که منجر به

precision کمتری می شود. بسته به کاربرد و الزامات خاص، تنظیم بیشتر مدل یا کاوش در الگوریتم‌های مختلف ممکن است برای بهبود عملکرد ضروری باشد.

همینجور که میبنید در واقع این نتیجه نشان داد که باید مدل را بر روی داده های *test* نیز ارزیابی کنیم.

توضیح کد :

```
# Convert probabilities to binary predictions (0 or 1)
y_pred_binary = np.where(y_pred_test >= 0.5, 1, 0)
```

در واقع با دستور *np.where(...)* ما داده هایی که بیشتر از *5.* باشند را رند کرده و برابر ۱ قرار میدهیم و اگر کم تر باشند برابر ۰ میشوند.

```
# Calculate accuracy
accuracy = np.mean(y_pred_binary == y_test)

# Calculate precision, recall, and F1-score
TP = np.sum((y_pred_binary == 1) & (y_test == 1))
FP = np.sum((y_pred_binary == 1) & (y_test == 0))
FN = np.sum((y_pred_binary == 0) & (y_test == 1))

precision = TP / (TP + FP)
recall = TP / (TP + FN)
f1_score = 2 * precision * recall / (precision + recall)
```

در اینجا نیز برای بدست آوردن *TP,...* از دستورات شرطی رفتیم مثلا تعداد داده هایی که *y_pred_binary* برابر ۱ و *y_test* برابر ۱ باشند را با هم جمع میکنیم.(در واقع تعداد داده های ۲ آرایه را که برابر ۱ باشند را حساب میکنیم.)فرمول هر کدام هم میتوان در بالا مشاهده کرد.

۴)

بعد از *import* کردن کتابخانه های مورد نیاز کد بدین شکل است :

```
# Train logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict probabilities and binary predictions on test data
y_pred_binary = model.predict(X_test)
```

مدل ما بر روی داده های آموزش درست کرده سپس بر روی داده های تست ارزیابی کرده و سپس با معیار های بخش قبل آنهارا بدست می آوریم :

```
# Calculate performance metrics
```

```
accuracy = accuracy_score(y_test, y_pred_binary)
precision = precision_score(y_test, y_pred_binary)
recall = recall_score(y_test, y_pred_binary)
f1 = f1_score(y_test, y_pred_binary)
```

که نتیجه بدین شکل است :

```
Performance Metrics:
Accuracy: 0.475
Precision: 0.475
Recall: 1.0
F1-score: 0.6440677966101694
```

که دقیقا مانند نتیجه قبل میباشد.

اگر ما بر روی داده های آموزش مان ***preprocess*** انجام ندهیم(یعنی به جای ***X_train=X_train_standarized*** را قرار ندهیم) نتیجه بدین شکل میشود :

```
Performance Metrics:
Accuracy: 0.975
Precision: 1.0
Recall: 0.9473684210526315
F1-score: 0.972972972972973
```

این نتیجه در واقع وقتی بدست آمده است که ***preprocess*** انجام ندهیم.

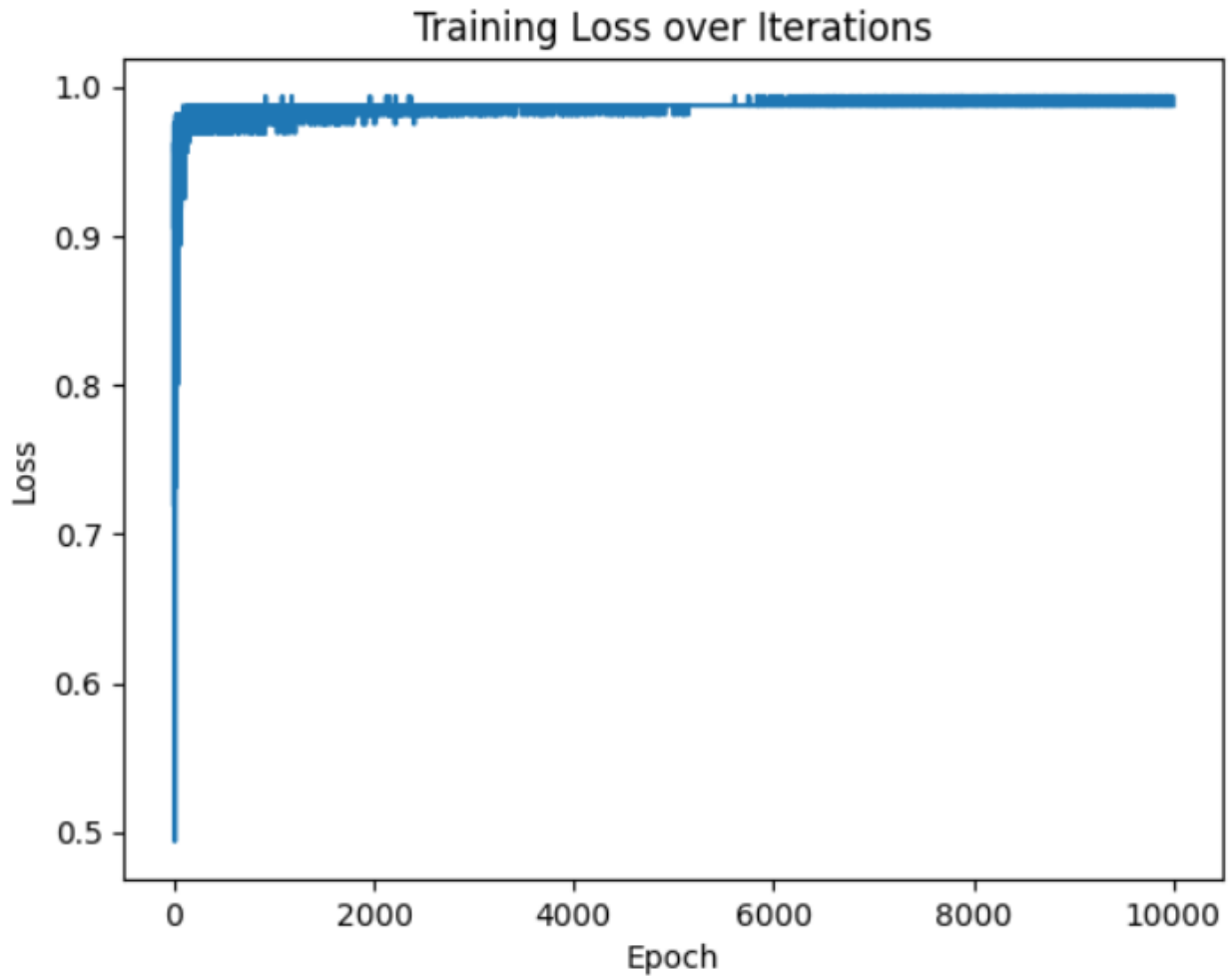
در واقع راه مستقیم برای ***logisticregression*** وجود ندارد ولی در ***SGD*** میتوان بدین صورت عمل کرد :

```
# Initialize SGDClassifier with 'log' loss
clf = SGDClassifier(loss='log_loss', max_iter=10000, tol=1e-3)

# Train the classifier
loss_values = []
for i in range(10000):  # Run for 10 epochs
    clf.partial_fit(X_train, y_train, classes=[0, 1])  # Specify the
classes explicitly
    loss_values.append(clf.score(X_train, y_train))  # Compute the loss
using the score method
```

درواقع ما در اینجا اول مدل خود را تعریف کرده سپس آنرا برای ۱۰۰۰۰ بار انجام داده و در داخل حلقه، این خط متد *partial_fit* شی طبقه‌بندی کننده *clf* را فراخوانی می‌کند. *partial_fit* برای یادگیری آنلاین یا مینی دسته ای استفاده می شود، جایی که طبقه بندی کننده به صورت تدریجی بر روی دسته های کوچک داده آموزش داده می شود. پارامترهای مدل را بر اساس داده های آموزشی ارائه شده (*X_train*، *y_train*) به روز می کند. پارامتر کلاس ها کلاس های منحصر به فرد را در مسئله طبقه بندی باینری مشخص می کند. این برای طبقه‌بندی‌کننده‌هایی مانند *SGDClassifier* لازم است تا به درستی مشکلات طبقه‌بندی باینری را مدیریت کنند.

سپس پس از هر بار تکرار آموزش، این خط *accuracy* مدل را بر روی داده های آموزشی فعلی (*X_train*، *y_train*) با استفاده از روش *score* محاسبه می کند. روش *score* میانگین دقت مدل را بر روی داده ها و برچسب های داده شده برمی گرداند. سپس دقت محاسبه شده به لیست *loss_values* اضافه می شود.

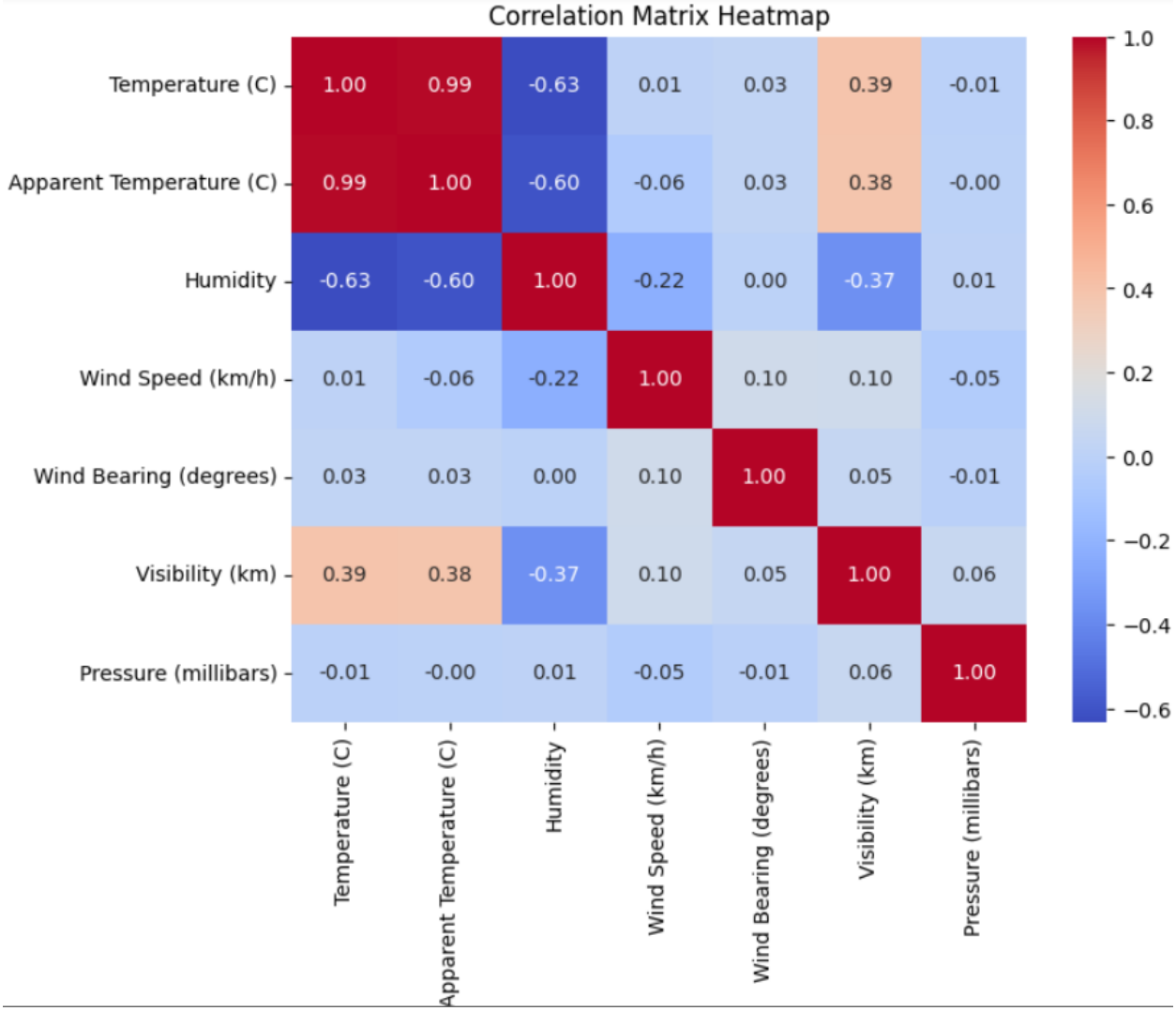
نتیجه بدین شکل میشود :

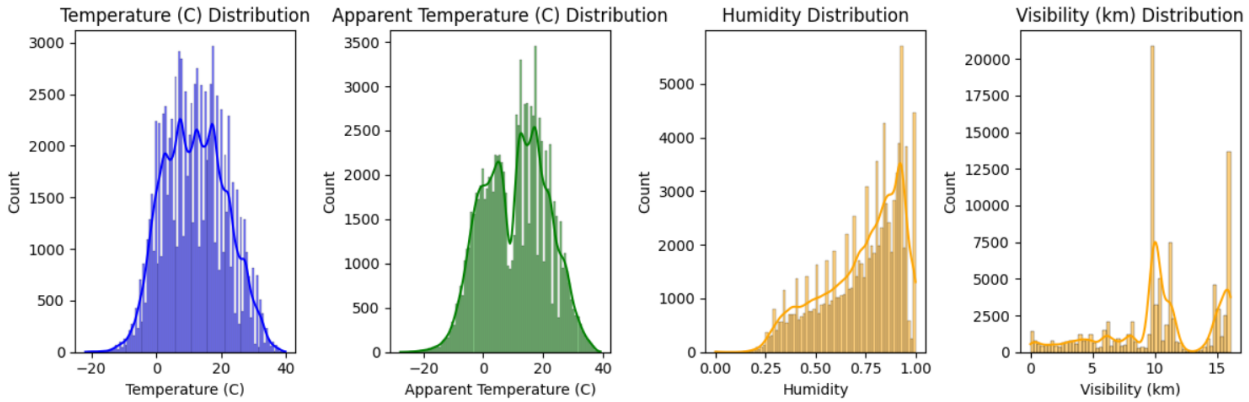


سوال ۳)

۱)

نتایج ماتریس *correlation* و نمودار *histogram* بدین شکل میباشد :





خوب همینجوری که ماتریس همبستگی را مشاهده میکنید بیشترین ارتباط با *Temperature,apparent* *temp* را *humidity* و *visibility* دارند. ارتباط بین *humidity* با ۲ ویژگی دیگر منفی میباشد به این دلیل که رابطه عکس بین آن با ۲ ویژگی دیگر میباشد یعنی مثلا با زیاد شدن *humidity* ۲ ویژگی دیگر کم میشوند.مورد بعدی که معلوم است این است که عدد همبستگی بین ۲ ویژگی دیگر زیاد بوده(*.99*)که نشان دهنده ارتباط زیاد بین این ۲ ویژگی میباشد.

در نمودار های پراکندگی دما مشاهده میکنید ۲ نمودار بیشتر در یک بازه دمایی وجود دارند.مورد بعدی این است که داده های نمودار *Temperature* به صورت *bell shape* میباشند که نشان دهنده این است که بیشتر داده ها در اطراف میانگین دمایی هستند.(*apparent temp* هم تقریبا بدین شکل میباشد ولی بدلیل افت داده در اطراف میانگین بدین شکل شده است و اینکه کمی به سمت راست مایل است.)در این ۲ ویژگی تنوع بیشتری در داده ها وجود دارد.

در *Humidity* همینطور که مشاهده میکنید داده ها به سمت راست تمایل دارند(*skewed to the right*) و این نشان دهنده این است که داده ها بیشتر در سمت راست میانگین داده ها هستند.این دقیقا بر عکس ۲ ویژگی دیگر میباشد که داده ها در اطراف میانگین داده ها تجمع داشتند.مورد بعدی اینکه در واقع وقتی داده ها به آخر بازه میرسند ۲ ویژگی دیگر کم شده ولی *humidity* بیشتر میشود.

در *visibility* نیز داده ها در ۲ جا بیشتر از جاهای دیگر وجود دارند.

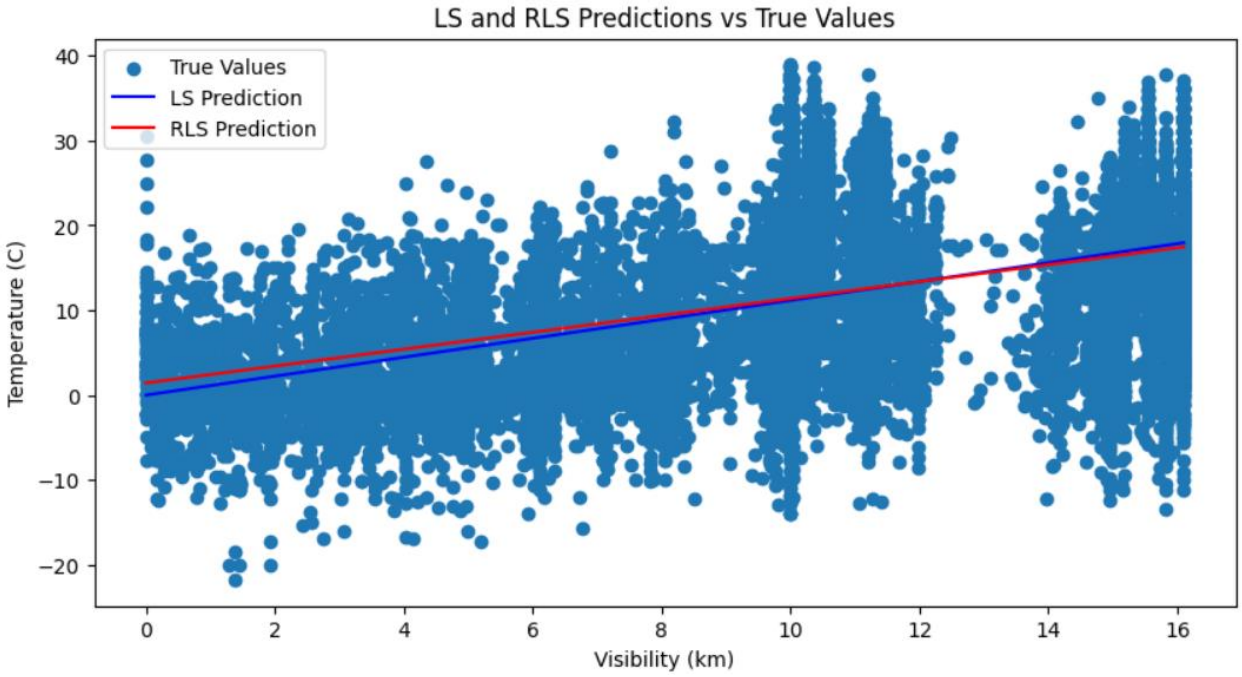
۲)

ما در اینجا میخواهیم *Temprature&Apparent Temp* را تخمین بزنیم.برای این منظور از *3 Feature* استفاده میکنیم.۱)*Humidity* ۲)*Visibility* ۳)*Wind Bearing (degrees)*

اول از همه نتیاج مدل هایی که برای تخمین *Temprature* اسفاده کردیم را نشان میدهیم :

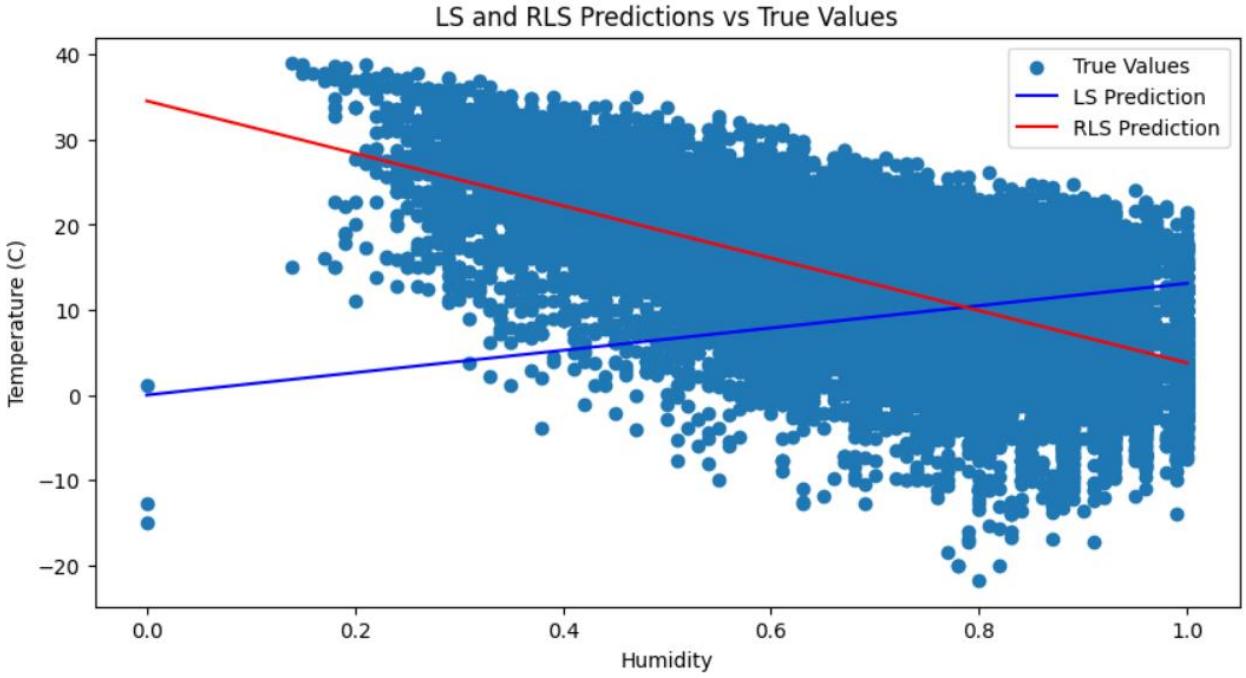
Visibility :

```
LS Mean Squared Error: 79.0675573175552
LS R-squared: 0.13990917051660678
<ipython-input-10-2ca861c908b4>:35: Deprec
  alpha = float((1 + x.T*z)**(-1))
<ipython-input-10-2ca861c908b4>:36: Deprec
  self.a_priori_error = float(t - self.w.T*
<ipython-input-10-2ca861c908b4>:37: Deprec
  self.w = self.w + (t-alpha*float(x.T*(sel
<ipython-input-10-2ca861c908b4>:64: Deprec
  return float(self.w.T*x)
RLS Mean Squared Error: 78.25098313096889
RLS R-squared: 0.14879180194348918
```



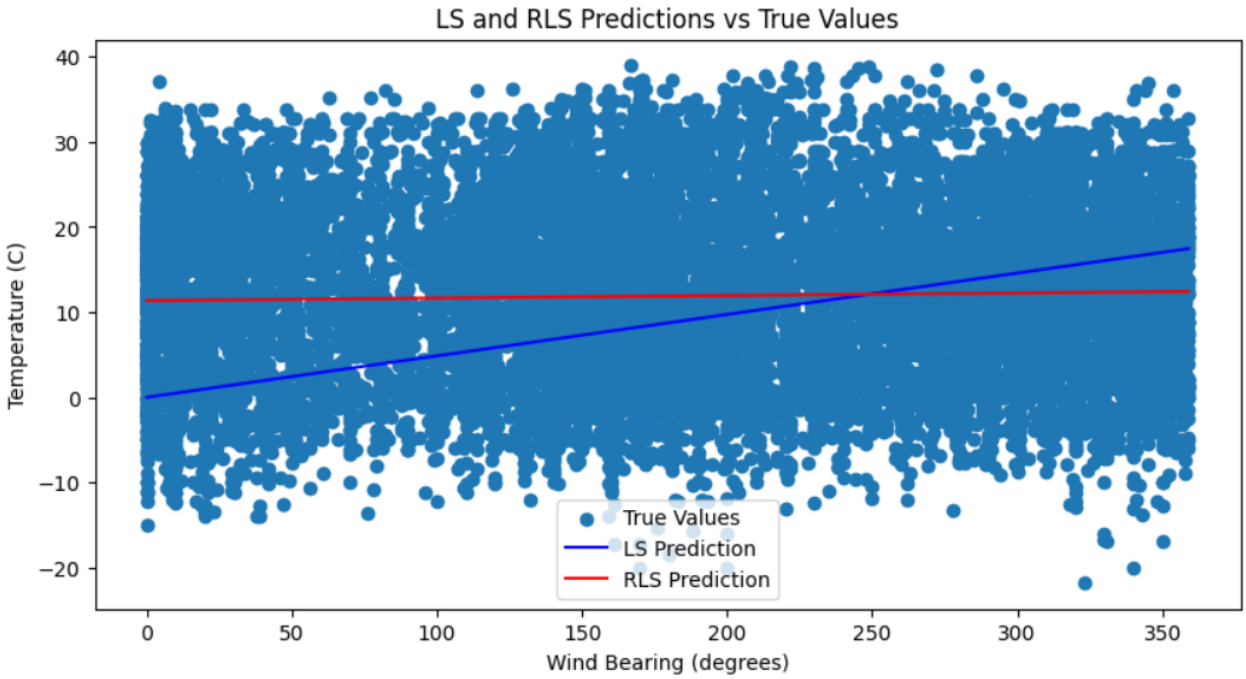
: *Humidity*

```
LS Mean Squared Error: 135.3640896416121
LS R-squared: -0.4724801940513328
<ipython-input-10-2ca861c908b4>:35: DeprecationWar
  alpha = float((1 + x.T*z)**(-1))
<ipython-input-10-2ca861c908b4>:36: DeprecationWar
  self.a_priori_error = float(t - self.w.T*x)
<ipython-input-10-2ca861c908b4>:37: DeprecationWar
  self.w = self.w + (t-alpha*float(x.T*(self.w+t*z
<ipython-input-10-2ca861c908b4>:64: DeprecationWar
  return float(self.w.T*x)
RLS Mean Squared Error: 54.46504011052992
RLS R-squared: 0.40753346738193597
```



: *Wind Bearing*

```
LS Mean Squared Error: 125.38845486045247
LS R-squared: -0.36396600334361007
<ipython-input-10-2ca861c908b4>:35: Deprecatio
  alpha = float((1 + x.T*z)**(-1))
<ipython-input-10-2ca861c908b4>:36: Deprecatio
  self.a_priori_error = float(t - self.w.T*x)
<ipython-input-10-2ca861c908b4>:37: Deprecatio
  self.w = self.w + (t-alpha*float(x.T*(self.w
<ipython-input-10-2ca861c908b4>:64: Deprecatio
  return float(self.w.T*x)
RLS Mean Squared Error: 91.9358653832539
RLS R-squared: -7.129851221510108e-05
```



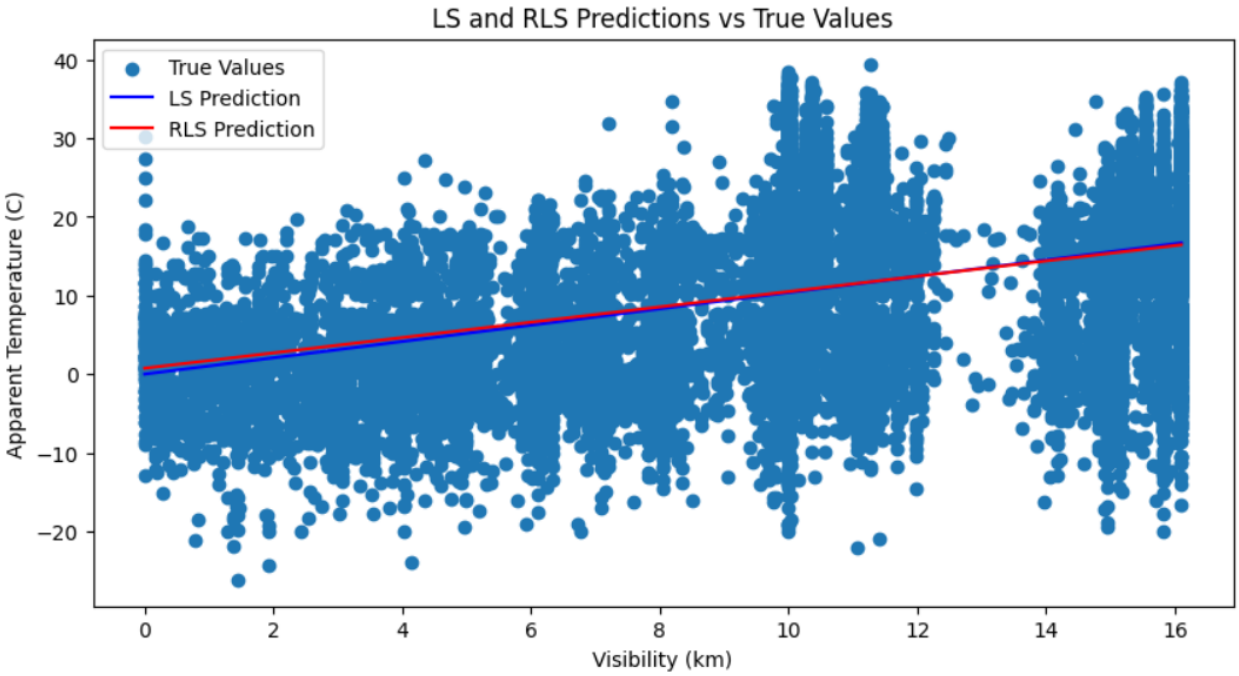
خوب همینطور که مشاهده میکنید بهترین میانگین خطا *LS* به ترتیب برای *Wind bearing , Visibility* و *Humidity* میباشد.دلیل آن این میتواند باشد که با اینکه قدر مطلق همبستگی *Humidity* بیشتر از بقیه میباشد ولی این موضوع که مقدار آن منفی است تاثیر زیادی بر روی دقت مدل آن دارد.۲ تای دیگه هم به ترتیب بزرگ بودن همبستگی هستند.

بهترین میانگین خطا برای *RLS* به ترتیب برای *Visibility , Humidity* و *Wind bearing* میباشد.این نشان دهنده این میباشد که بهترین مدل برای *RLS* ارتباط مستقیم با قدر مطلق مقدار میانگین خطا دارد.

برای *Apparent Temprature* :

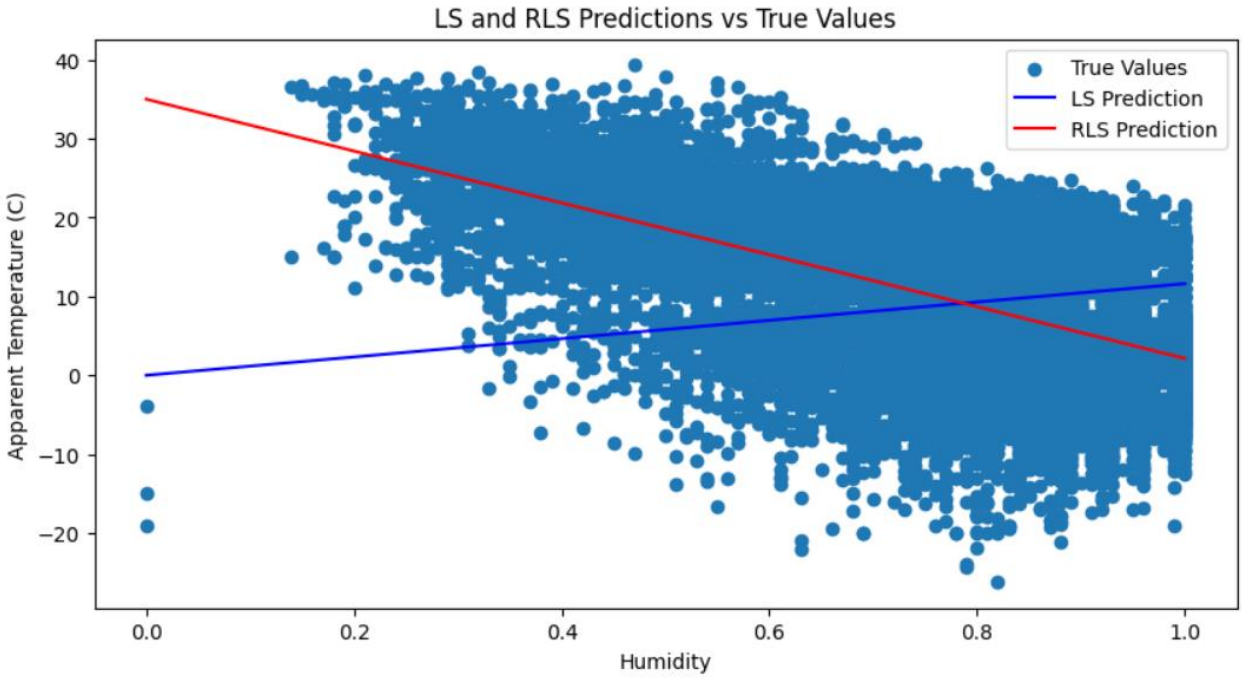
Visibility :

```
LS Mean Squared Error: 98.5903742889745
LS R-squared: 0.1419106077513249
<ipython-input-10-2ca861c908b4>:35: Deprecati
  alpha = float((1 + x.T*z)**(-1))
<ipython-input-10-2ca861c908b4>:36: Deprecati
  self.a_priori_error = float(t - self.w.T*x)
<ipython-input-10-2ca861c908b4>:37: Deprecati
  self.w = self.w + (t-alpha*float(x.T*(self.
<ipython-input-10-2ca861c908b4>:64: Deprecati
  return float(self.w.T*x)
RLS Mean Squared Error: 98.48300518873218
RLS R-squared: 0.14284510350344748
```



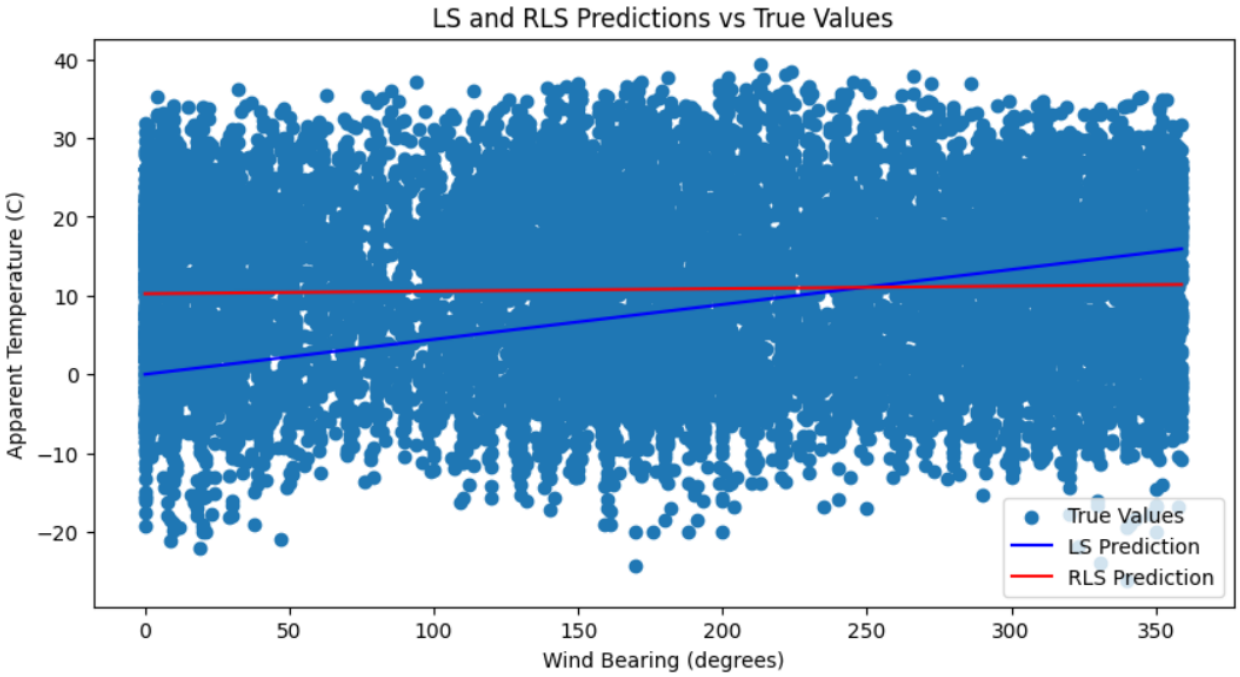
: *Humidity*

```
LS Mean Squared Error: 155.30419813873576
LS R-squared: -0.3517028001529634
<ipython-input-10-2ca861c908b4>:35: Deprecat
  alpha = float((1 + x.T*z)**(-1))
<ipython-input-10-2ca861c908b4>:36: Deprecat
  self.a_priori_error = float(t - self.w.T*x
<ipython-input-10-2ca861c908b4>:37: Deprecat
  self.w = self.w + (t-alpha*float(x.T*(self
<ipython-input-10-2ca861c908b4>:64: Deprecat
  return float(self.w.T*x)
RLS Mean Squared Error: 72.21762535858043
RLS R-squared: 0.37144798667716505
```



: *Wind Bearing*

```
LS Mean Squared Error: 142.3900997677759
LS R-squared: -0.23930388796204172
<ipython-input-10-2ca861c908b4>:35: DeprecationWarr
  alpha = float((1 + x.T*z)**(-1))
<ipython-input-10-2ca861c908b4>:36: DeprecationWarr
  self.a_priori_error = float(t - self.w.T*x)
<ipython-input-10-2ca861c908b4>:37: DeprecationWarr
  self.w = self.w + (t-alpha*float(x.T*(self.w+t*z)
<ipython-input-10-2ca861c908b4>:64: DeprecationWarr
  return float(self.w.T*x)
RLS Mean Squared Error: 114.90579855292775
RLS R-squared: -9.2022747873565 79e-05
```



نتایج مانند *Temprature* میباشد.(این میتواند به این دلیل باشد که همبستگی زیادی بین *Temp&apparent Temp* میباشد.)

۳)

حداقل مربعات وزنی (*WLS*) گونه ای از روش حداقل مربعات معمولی (*OLS*) است که در رگرسیون خطی استفاده می شود. به ویژه زمانی مفید است که فرض واریانس ثابت خطاها نقض شود، به این معنی که تغییرپذیری خطاها در تمام سطوح متغیرهای مستقل ثابت نیست.

در *WLS*، به هر مشاهده وزنی داده می شود که نشان دهنده قابلیت اطمینان یا دقت آن است. به مشاهداتی که پایایی بالاتری دارند، وزن‌های بالاتری نسبت داده می‌شوند، در حالی که به مشاهداتی که پایایی کمتری دارند، وزن‌های کمتری اختصاص می‌دهند. وزن ها معمولاً به طور معکوس متناسب با واریانس خطاها انتخاب می شوند.

هدف *WLS* به حداقل رساندن مجموع مجذورهای باقیمانده وزن شده، به جای مجموع مجذور باقیمانده ها در *OLS* است. از نظر ریاضی، برآوردگر حداقل مربعات وزنی را می توان به صورت زیر بیان کرد:

$$Q = \begin{pmatrix} Q_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & Q_{NN} \end{pmatrix}$$

Q در واقع ماتریس قطری وزن ها میباشد.

$$Q = \frac{1}{var(y)} \rightarrow \theta = (X^T Q X)^{-1} X^T Q y$$

انتخاب وزن ها به مشکل خاص و ویژگی های داده ها بستگی دارد. طرح های وزن دهی رایج شامل استفاده از معکوس واریانس باقیمانده ها یا استفاده از تابعی از باقیمانده های مطلق است.

WLS به ویژه در شرایطی که ناهمسانی وجود دارد مفید است، زیرا می‌تواند تخمین‌های کارآمدتری از ضرایب را در مقایسه با OLS با دادن وزن بیشتر به مشاهدات قابل اعتمادتر و وزن کمتر به مشاهدات کمتر قابل اعتماد ارائه دهد.

اول از همه نتیاج مدل هایی که برای تخمین ***Temprature*** اسفاده کردیم را نشان میدهیم :

Visibility :

```
                            WLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.154
Model:                            WLS   Adj. R-squared:                  0.154
Method:                 Least Squares   F-statistic:                 1.760e+04
Date:                Tue, 09 Apr 2024   Prob (F-statistic):               0.00
Time:                        22:06:53   Log-Likelihood:            -3.4644e+05
No. Observations:               96453   AIC:                         6.929e+05
Df Residuals:                   96451   BIC:                         6.929e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.6710      0.075     35.461      0.000       2.523       2.819
x1             0.8951      0.007    132.671      0.000       0.882       0.908
==============================================================================
Omnibus:                      778.145   Durbin-Watson:                   0.075
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              698.913
Skew:                           0.165   Prob(JB):                    1.71e-152
Kurtosis:                       2.744   Cond. No.                         29.9
==============================================================================
```

Humidity :

```
                            WLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.400
Model:                            WLS   Adj. R-squared:                  0.400
Method:                 Least Squares   F-statistic:                 6.423e+04
Date:                Tue, 09 Apr 2024   Prob (F-statistic):               0.00
Time:                        22:07:39   Log-Likelihood:            -3.2991e+05
No. Observations:               96453   AIC:                         6.598e+05
Df Residuals:                   96451   BIC:                         6.598e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         34.6369      0.093    373.651      0.000      34.455      34.819
x1           -30.8944      0.122   -253.442      0.000     -31.133     -30.655
==============================================================================
Omnibus:                     2385.781   Durbin-Watson:                   0.043
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             2566.298
Skew:                          -0.394   Prob(JB):                         0.00
Kurtosis:                       3.131   Cond. No.                         7.95
==============================================================================
```

: Wind Bearing

```
                            WLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.001
Model:                            WLS   Adj. R-squared:                  0.001
Method:                 Least Squares   F-statistic:                     86.82
Date:                Tue, 09 Apr 2024   Prob (F-statistic):           1.22e-20
Time:                        22:12:03   Log-Likelihood:            -3.5448e+05
No. Observations:               96453   AIC:                         7.090e+05
Df Residuals:                   96451   BIC:                         7.090e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         11.4325      0.062    184.816      0.000      11.311      11.554
x1             0.0027      0.000      9.317      0.000       0.002       0.003
==============================================================================
Omnibus:                     2806.209   Durbin-Watson:                   0.027
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1449.767
Skew:                           0.098   Prob(JB):                         0.00
Kurtosis:                       2.432   Cond. No.                         435.
==============================================================================
```

همینطور که میبینید بهترین مدل Humidity و بعد از آن Visibility و Wind bearing میباشد.(R-squered) این مدل از مدل RLS بهتر نمیباشد ولی ۲ تای دیگه از مدل RLS خود بهتر میباشند.

برای ***Apparent Temprature*** :

Visibility :

```
                            WLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.146
Model:                            WLS   Adj. R-squared:                  0.146
Method:                 Least Squares   F-statistic:                 1.645e+04
Date:                Tue, 09 Apr 2024   Prob (F-statistic):               0.00
Time:                        22:34:39   Log-Likelihood:            -3.5785e+05
No. Observations:               96453   AIC:                         7.157e+05
Df Residuals:                   96451   BIC:                         7.157e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.7766      0.085      9.160      0.000       0.610       0.943
x1             0.9740      0.008    128.261      0.000       0.959       0.989
==============================================================================
Omnibus:                      782.154   Durbin-Watson:                   0.076
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              532.156
Skew:                          -0.042   Prob(JB):                    2.78e-116
Kurtosis:                       2.646   Cond. No.                         29.9
==============================================================================
```

Humidity :

```
                            WLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.363
Model:                            WLS   Adj. R-squared:                  0.363
Method:                 Least Squares   F-statistic:                 5.499e+04
Date:                Tue, 09 Apr 2024   Prob (F-statistic):               0.00
Time:                        22:35:05   Log-Likelihood:            -3.4369e+05
No. Observations:               96453   AIC:                         6.874e+05
Df Residuals:                   96451   BIC:                         6.874e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         35.0879      0.107    328.119      0.000      34.878      35.298
x1           -32.9745      0.141   -234.489      0.000     -33.250     -32.699
==============================================================================
Omnibus:                     3365.837   Durbin-Watson:                   0.048
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             3727.020
Skew:                          -0.476   Prob(JB):                         0.00
Kurtosis:                       3.144   Cond. No.                         7.95
==============================================================================
```

: Wind Bearing

```
                            WLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.001
Model:                            WLS   Adj. R-squared:                  0.001
Method:                 Least Squares   F-statistic:                     81.35
Date:                Tue, 09 Apr 2024   Prob (F-statistic):           1.92e-19
Time:                        22:35:42   Log-Likelihood:            -3.6541e+05
No. Observations:               96453   AIC:                         7.308e+05
Df Residuals:                   96451   BIC:                         7.308e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         10.3128      0.069    148.860      0.000      10.177      10.449
x1             0.0029      0.000      9.020      0.000       0.002       0.004
==============================================================================
Omnibus:                     5385.148   Durbin-Watson:                   0.029
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             2070.134
Skew:                          -0.054   Prob(JB):                         0.00
Kurtosis:                       2.290   Cond. No.                         435.
==============================================================================
```

نتایج این قسمت هم مانند قبل میباشد.

گزارش کد :

برای بدست آوردن correlation matrix بدین شکل عمل کردیم :

```
# Calculate the correlation matrix
correlation_matrix = weather_data[['Temperature (C)', 'Apparent
Temperature (C)', 'Humidity' ,'Wind Speed (km/h)','Wind Bearing
(degrees)','Visibility (km)','Pressure (millibars)' ]].corr()
```

و سپس بدین شکل آنرا نمایش دادیم :

```
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
```

برای کشیدن هیستوگرام پراکندگی بدین شکل عمل کردیم :

```
plt.subplot(1, 4, 1)
sns.histplot(weather_data['Temperature (C)'], kde=True, color='blue')
```

۳ تای دیگر هم مانند این میباشد فقط رنگ های آنها با هم متفاوت است.

کد RLS همان کد توضیح داده شده در کلاس TA میباشد. توضیح این کد بدین شکل است :

این مقدار پارامتر num_vars نشان‌دهنده تعداد متغیرها از جمله ثابت در مدل رگرسیونی است.

self.A = delta*np.matrix(np.identity(self.num_vars)): این ماتریس A را به صورت یک ماتریس مورب با ابعاد num_vars x num_vars مقداردهی می‌کند، که در آن عناصر مورب روی دلتا ضربدر ماتریس هویت (identity matrix)تنظیم می‌شوند.

self.w = np.matrix(np.zeros(self.num_vars)): این بردار وزن w را به عنوان بردار ستونی از صفرها با عناصر num_vars مقداردهی اولیه می کند.

self.w = self.w.reshape(self.w.shape[1],1): این بردار وزن w را به یک بردار ستونی با ردیف num_vars و 1 ستون تغییر شکل می دهد.

self.lam_inv = lam**(-1): معکوس forgetting factor (lam) را محاسبه کرده و به متغیر نمونه self.lam_inv نسبت می دهد.

self.sqrt_lam_inv = math.sqrt(self.lam_inv): این جذر معکوس ضریب فراموشی lam را با استفاده از تابع sqrt از ماژول ریاضی محاسبه می کند و آن را به متغیر نمونه self.sqrt_lam_inv اختصاص می دهد.

self.a_priori_error = 0: این متغیر نمونه self.a_priori_error را به 0 مقداردهی می کند، که برای ذخیره خطای پیشینی (آنی) استفاده می شود.

self.num_obs = 0: این متغیر نمونه self.num_obs را به 0 مقداردهی می کند، که برای پیگیری تعداد مشاهدات اضافه شده به مدل استفاده می شود.

بقیه کد چندین روش (add_obs و fit و get_error و predict) را برای کلاس RLS تعریف می کند که عملکرد الگوریتم RLS را پیاده سازی می کند. این روش ها به ترتیب برای افزودن مشاهدات به مدل، fit کردن مدل، محاسبه خطا و پیش بینی استفاده می شوند.

add_obs(self, x, t): این متد وظیفه اضافه کردن یک مشاهده (x) با برچسب (t) مربوط به آن را به مدل دارد. این پارامترهای مدل را بر اساس مشاهدات جدید به روز می کند.

x یک بردار ستونی است که به صورت یک ماتریس numpy نمایش داده می شود. این شامل متغیرهای مستقل (ویژگی های) مشاهده است.

t یک اسکالر واقعی است که نشان دهنده متغیر هدف (برچسب) مربوط به مشاهده است.

داخل روش:

z مقدار میانی self.lam_inv * self.A * x را محاسبه می کند که در الگوریتم RLS استفاده می شود.

آلفا ضریب به روز رسانی را با استفاده از فرمول محاسبه می کند :

$$(1 + x^T z)^{-1}$$

self.a_priori_error خطای پیشینی (آنی) را ذخیره می کند که به عنوان تفاوت بین هدف مشاهده شده t و هدف پیش بینی شده با استفاده از پارامترهای مدل فعلی self.w محاسبه می شود.

پارامترهای مدل self.w و self.A بر اساس مشاهده جدید و ضریب آلفای به روز رسانی به روز می شوند.

self.num_obs 1 افزایش می یابد تا تعداد مشاهدات اضافه شده به مدل را پیگیری کند.

fit(self, X, y): این روش مدل RLS را با داده های آموزشی X و y مطابقت می دهد.

X یک آرایه numpy حاوی متغیرهای مستقل (ویژگی) داده های آموزشی است. هر مشاهده باید یک ردیف در X باشد و یک ضریب ثابت باید برای هر مشاهده اضافه شود.

y یک آرایه numpy است که شامل متغیرهای هدف (برچسب) مربوط به داده های آموزشی است.

داخل روش:

بر روی هر مشاهده در داده های آموزشی تکرار می شود.

برای هر مشاهده، بردار ویژگی X[i] را جابجا می کند، آن را به یک ماتریس numpy تبدیل می کند و متد add_obs را فراخوانی می کند تا مشاهده را به مدل اضافه کند.

get_error(self): این متد خطای پیشینی (آنی) مدل را برمی گرداند.

داخل روش:

به سادگی مقدار self.a_priori_error را که در روش add_obs محاسبه شده است، برمی گرداند.

predict(self, x): این روش متغیر هدف را برای مشاهدات x پیش بینی می کند.

x یک ماتریس ناقص است که نشان دهنده بردار ویژگی مشاهداتی است که پیش بینی برای آن انجام شده است.

داخل روش:

متغیر هدف پیش بینی شده را با ضرب جابه جایی بردار وزن self.w با بردار ویژگی x محاسبه می کند و نتیجه را به صورت شناور برمی گرداند.

توضیح کد قسمت LS and RLS :

```
# Least Squares (LS) estimation
ls_theta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train

# Make predictions
ls_predictions = np.dot(X_test, ls_theta)
```

در واقع در این ۲ خط اول از همه ضرایب را بدست آورده سپس در خط بعدی predict رو بر روی ضرایب و داده های test انجام داده است.

```
# Calculate mean squared error for LS and RLS
ls_mse = mean_squared_error(y_test, ls_predictions)

print("LS Mean Squared Error:", ls_mse)

r_squared = r2_score(y_test,  ls_predictions)
print("LS R-squared:", r_squared)
```

در اینجا هم mean squared error و r2 score را بدست آورده ایم.

سپس پارامتر های RLS را معلوم میکنیم و بعد از آن بدین شکل :

```
for i in range(len(X_train)):
    x = np.insert(X_train[i], 0, 1)  # Prepend 1 for constant coefficient
    rls_model.add_obs(x.reshape(-1, 1), y_train[i][0])  # Assuming y_train
is a single target
```

for i in range(len(X_train)):: این حلقه بر روی هر شاخص i در محدوده طول مجموعه آموزشی (X_train) تکرار می شود و به ما امکان دسترسی به هر مشاهده را می دهد.

x = np.insert(X_train[i], 0, 1): برای هر مشاهده X_train[i]، این خط مقدار ثابت 1 را در ابتدای بردار ویژگی درج می کند. این ثابت برای محاسبه عبارت وقفه در مدل رگرسیون خطی اضافه می شود. پس از درج ثابت، x تبدیل به بردار ویژگی افزوده می شود.

rls_model.add_obs(x.reshape(-1, 1), y_train[i][0]): این خط بردار ویژگی افزوده شده x و مقدار هدف متناظر آن y_train[i][0] را با استفاده از add_obs به مدل RLS اضافه می‌کند. روش.

x.reshape(-1, 1): بردار ویژگی افزوده شده x را به بردار ستونی تغییر شکل می دهد، که توسط متد add_obs لازم است.

y_train[i][0] : به مقدار هدف برای مشاهده فعلی i در مجموعه آموزشی دسترسی پیدا می کند. فرض می‌کند که y_train یک آرایه دو بعدی است که در آن هر ردیف یک مقدار هدف واحد را نشان می‌دهد. [0] برای استخراج مقدار هدف از آرایه استفاده می شود، با فرض اینکه یک هدف واحد باشد.

سپس predictکرده و نتایج را print میکنیم و در آخر plotمیکنیم:

```
sorted_indices = np.argsort(X_test.flatten())
X_test_sorted = X_test[sorted_indices]
ls_predictions_sorted = ls_predictions[sorted_indices]
rls_predictions_sorted = rls_predictions[sorted_indices]
plt.plot(X_test_sorted, ls_predictions_sorted, color='blue', label='LS
Prediction')
plt.plot(X_test_sorted, rls_predictions_sorted, color='red', label='RLS
Prediction')
```

نکته ای که برای کشیدن خط وجود دارد این است که داده ها را به صورت صعودی مرتب کرده و سپس predict را انجام داده و خط آنرا Plot میکنیم.