Data loading & EDA
Seismic Event Classification (Low-Magnitude, M1–4)

Bu model calssifcationga asoslangan sesmik holatni tekshiradigan proyekt ekan. :

```
DATA_PATH = Path("data/raw/usgs_m1_4_aug-nov_2025.csv")

if not DATA_PATH.exists():
    raise FileNotFoundError(
        f"Dataset not found at {DATA_PATH.resolve()}\n"
        "Place the CSV at data/raw/ or update DATA_PATH. "
        "See data/README.md for how to reproduce the dataset."
    )

df = pd.read_csv(DATA_PATH)
df.head()
```

Bunda biz odatda datatestni : **df = pd.read_csv('relative path')** orqali chaqirar edik. Ammo bu yerda **DATA_PATH = Path("fayl yo'li")** bu yerda asosiy yangilik men uchun **Path()** bu maxsus obyekt bo'lib unda **exist(), resolve()** kabi metodlar borligida.

if not DATA_PATH.exist() --> bu yerda esa agar fayl yoki papka mavjudmi degan sovlga javob beradi. **raise** FileNotFoundError esa bu dataset topilmadi -- dasturni to'xtat va xato chiqar degani ekan. **DATA_PATH.resolve()** --> bu esa Pathni to'liq absolute yo'lga aylantirar ekan masalan: data/raw/file.csv bo'lsa resolve() orqali esa: /home/user/project/data/raw/file.csv kompyuterdagi to'liq manzilni berar ekan.

```
# finding quick stats for non-numeric columns
df[df.columns[df.dtypes == "object"]].describe(include="all")
```

Bu yerda ham bir yangilk bo'ldi ya'ni object ustunlar olib ular haqida statistik ma'lumot berar ekan. **df- bu dataframe, df.columns-**bu esa hamma columnlarni olsa uni ichidagi **df.dtypes == 'object'** esa o'sha hamma ustunlar ichidan esa faqat **'object'**lisinigina ol va **describe(inlcude='all')** bu esa o'zi numeric ustunlar uchun ishlaydi ammo i**nclude='all'** berish orqali esa biz **caegoricalga** ham olishimiz mukin ekan.

```
df.isna().sum()
```

Bu ham farqli bo'ld. **df.insull().sum()** bilan **df.isna().sum()** ham bir hil vazifani bajarishi isna() esa zamonaviyroq ham ekanligini bildim.

```
# drop fields
dropped_fields = ["locationSource","magSource","updated", "id", "status"]
df.drop(dropped_fields, axis =1, inplace = True)
```

Bu yerda esa biz oldin **df.drop('', '', axis=1, inplace=True)** deya bitta qator kod bilan berib ketar edik. Ammo alohida bitta list qilib uning ichiga drop qiladigan ustunlarni belgilab olib tashlasak ham bo'lar ekan.

```
# time field correction
df["time"] = pd.to_datetime(df["time"])
# ensure correct field type correction
df.info()
```

Bu ham yangilik bo'ldi sababi datasetimizda time bu oddiy **string**(**object**) bo'ladi bu orqali biz sanani solishtira olmaymiz, oy bo'yicha filtr qila olmaymiz va grafik chiza olmaymiz. pd.to_datetime() orqali esa biz u bilan ishlay olamiz masalan; **df['time'].dt.year, df['time'].dt.month, df['time'].dt.day** deya.

```
df = df[df['place'].str.contains(r'\d+\s*km.*?of', na=False)]
df.info()
```

Buni ham ko’rib ketdim sababi bunda har bir ustun ichdiagi har bir satrni tekshiradi. Masalan place ustuni ichidagi har bir satrni tekshirib (**str.contains()**) va uning ichidagi regex(matrdan aniq ushlash uchun maxsus til) ya’ni raw string - bu Python ichdiagi maxsus belgilarni o’zgarirmasdan qabul qil degani ekan:

\d+ → 1 yoki ko‘p raqam (10, 5, 123)
\s* → bo‘sh joy (ixtiyoriy)
km → aynan "km"
.*? → har qanday belgilar
of → aynan "of"

na=False esa agar palce ustinda **NaN** qiymat bo’lsa default error chiqarishi mumkin ekan shuni oldini olish maqsadida agar **NaN** bo’lsa **False** deb ket degani ekan.

```
df[["distance_to_city", "state"]] = df["place"].str.extract(r"(\d+(?:\.\d+)?)\s*km.*?,\s*(.*)")
```

str.extract() -- regex ichidagi qavslar yordamida ma’lumotlarni ajratib oladi.

(\d+(?:\.\d+)?) → boshida raqam (10 yoki 10.5)
\s* → bo‘sh joy
km → "km"
.*?, → vergulgacha bo‘lgan qism
\s* → bo‘sh joy
(.*) → verguldan keyingi hamma narsa (state)

```
# dropping unnecessary fields
df.reset_index(drop=True)
df.drop(["time", "place", "year"], axis=1, inplace=True)
df.info()
```

reset_index(drop=True) - bu indeksni boshidan qayta tartiblaydi, va pastagi kodda esa tashlab yuboriladigan ustunlarni belgilab olganin ko’rdik.

```
df['state'].value_counts()
```

Har bir qiymat nechchi marta takrorlanganini hisoblaydi. Y’ani state usunidagi qiymatlar nechchi martadan takrorlangan bunda eng ko’pi tartib bilan

```
# for data quality purposes, keep all states to follow the same naming convention
state_mapping = {"CA": "California", "NV": "Nevada"}

df['state'] = df['state'].replace(state_mapping)
df["state"].value_counts()
```

ketadi.

Bunda esa biz istlagan ustundagi biz uchun tushunarsiz bo’lgan qiymatlarni lug’at orqali agar CA ko’rsang bizga California qiliib ber agar NV ko’rsang bizga Nevada qiliib ber deydi. Va **replace()** orqali bizga shuni almashtirib ber deyiladi.

```
# replace nulls in nst with 1
df["nst"] = df["nst"].fillna(1)

# replace nulls in magNst with 1
df["magNst"] = df["magNst"].fillna(1)
```

Bu yerda esa fixed qiymat bilan to’ldriayapti. Agar shu nst va magNst ustunlarida NaN value bo’lsa ularni 1 deya.

```python
# drop rows with nulls in rms
df = df[df["rms"].notna()]

# drop rows with nulls in depthError
df = df[df["depthError"].notna()]

# Reset index after dropping rows
df = df.reset_index(drop=True)
```

Bu yerda esa shbu qatorlarni qanday True va False tekshirishini o'rgandim **notna()** orqali qilingan masking bilan ham drop qilib yuborsa bo'lar ekan. notna() bizda to'g'ridan-to'g'ri drop() ni bermadi ammo u bizga True va Falseni qaytaradi va bu df[mask] bo'gani hisobiga bizda boolean indexing bo'lib faqat Truelar qoladi Falselar esa droped amalga oshadi.

```python
# Reset index after dropping rows
df = df.reset_index(drop=True)
```

Va pastidan esa drop qilingan rowlarni qayta indexlab qo'yildi.

```python
# handling nulls in gap, dmin, horizontalError, magError by replacing nulls with the median of each field
nulls_col= ["gap", "dmin", "horizontalError", "magError"]#, "rms", "depthError"]

for col in nulls_col:
    # Compute median per event type
    col_median = df[col].median()
    # Fill nulls with the corresponding median
    df[col] = df[col].fillna(col_median)
    print(f"{col} nulls handled {col_median}")
```

Bu yerda esa yana ro'yxat shaklida utunalrni berdi va for loop orqali biroz biznikidan boshqacharoq yo'l bilan **median()** --ya'ni o'rta qiymati bilan to'ldirildi. Bu yerda alohida **col_median** -deya o'zgaruvchi va uni esa **df[col].median()** berib saqalb olib **df[col] = df[col].fillna(col_median)** to'ldirib qo'ya qoldi. Eng pastida esa debuggin uchun **print()** orqali shu 4 ta ustunni ko'rsatib berdi ya'ni bu ustun bu qiymat bilan to'ldirildi deya.

```python
# Select numeric columns and add the target/class column
pair_plot_df = df.select_dtypes(include=['int', 'float']).columns.tolist()+["type"]
```

Bu yerda esa biz oldin **int64** yoki **float64** deya berar edik bu yerda **int** va **float** deya berish va **tolist()** yani ro'yxatini qilib berishi va unga **+ ['example']** istalgan bir ustuni ro'yxatga qo'shsa bo'lishni ko'rdim.

```python
# Predictor & Target separation

y = df["type"]
X = df.drop(["type"], axis = 1)
X = pd.get_dummies(X, columns=["state_grouped", "net", "weekday", "magType"], drop_first=True)
print("Feature matrix shape:", X.shape)
print("Feature matrix shape:", y.shape)
```

Bu yerda esa biz ustunlar ichidan target ustunini tashlab keyin taget sizfatida alohida saqlab olish va one-hot encoding qilishni boshqacha versiyasini ko'rdim. Ya'ni X orqali barcha qolgan ustunlarga columns=[' ', ' ',..] qo'shishni va drop_first=True (bitta categoriyani asos sifdatida qoldirib qolganlarini esa 0/1 ko'rinishida ifodalash uchun kera eka. Bu esa bizga (**Multicollinearity**-bu feature'lar (X ustunlar) bir-biriga kuchli bog'liq bo'lib qolishi oldini olish uchun ekan).

```
# First split
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42, stratify=y)

# Second split
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
classes = np.unique(y_train)

print("Train:", X_train.shape, "Val:", X_val.shape, "Test:", X_test.shape)
```

Bu yerda esa data 3 ga bo'linanligi **Train, Validation va Test** uchun bo'lib olindi bu yerdagi logica shundan iboratki eng birinchi **X_train** va **y_train** uchun datani **60**% olinib qolgan **40**% esa yana ikkiga bo'lindi ular validation (o'rganish guruhi train va testdan alohida qism, u modelni o'rtgatib bo'lganimizdan keyin uni parametrlarini sozlash uchun ishlatiladi) va testga. Va stratifiy=y degan yangilk bo'ldi mega buning muhim jihati bu targetga bo'glangan holda train va temp ichidagi original target taqsimoti saqlansiz degani ekan. Ikkinchi splitda esa X_temp va y_temp dan olib uni ikki qismga bo'ldi bu X_val, x_test va y_val, y_testga teng yarimga 40%/2 = 20% dan bo'lib bergan. Va stratify=y_temp bo'lgani, sababi har bir split o'ziga tegishli bo'lgan target bilan stratify qilinadi. Shu bilan har bir qismda class balans saqlanadi.