machine learning cheat sheet

① Data analysis

(i) from ydata_profiling import ProfileReport, p = ProfileReport(df), p.to_notebook_iframe()

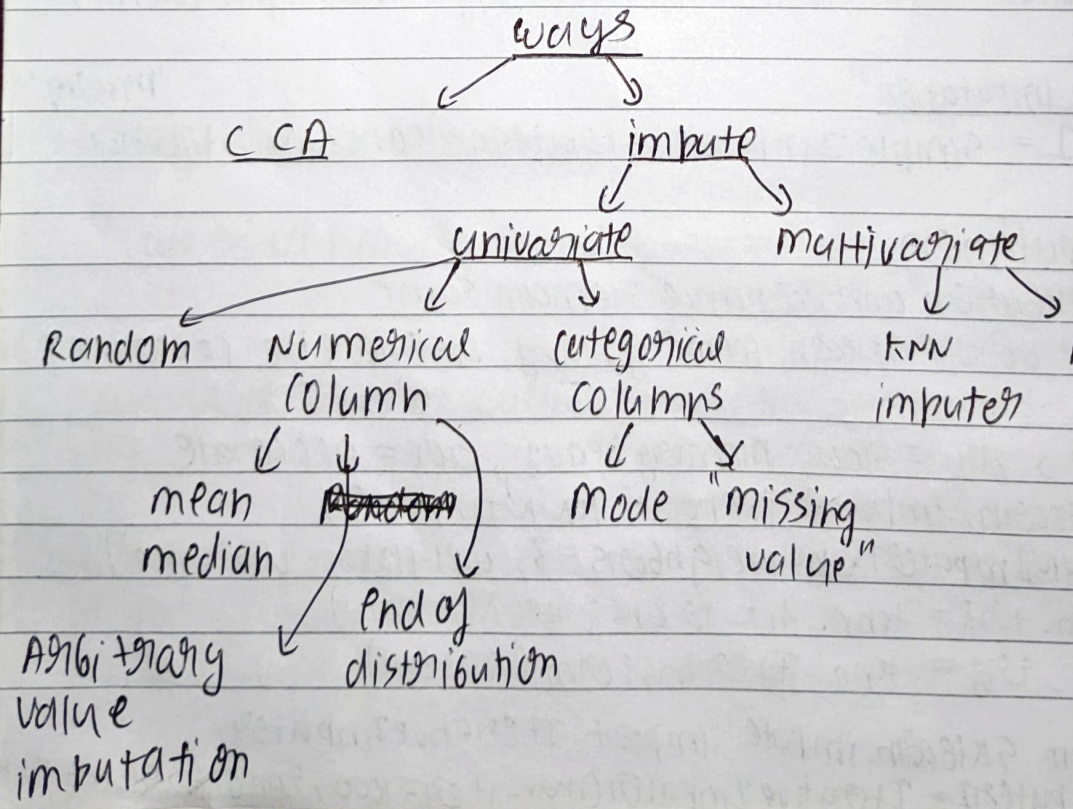
(ii) plt.scatter(c1, c2) (iii) lineplot = plt.plot(c1, c2) (iv) plt.bar(c1, c2) (v) sns.catplot(c1, c2, data, kind = 'bar'/'box'/'count') (vi) kde = sns.kdeplot(c1) (vii) stats.probplot(data, dist = "norm", plot = plt)

↳ from scipy import stats

② Feature Engineering

steps = Feature selection → missing value imputation → outlier detection → Feature scaling → encoding of categorical column → encoding of numerical column → transformation of columns → PCA

③ missing value imputation



→ x_train['cl_trg'] = x_train['cl']
→ Now apply transformation on cl_trg

we apply CCA and univariate when data missing is less then 5% and we have to do analysis after CCA and univariate imputation on the column.

CCA ↳ df.isnull().mean()*100
↳ new_df = df[cols].dropna()
→ Production code model can not handle missing value

mean median
adv → change in → distribution, co-relation, addition of outliers
imputer = SimpleImputer(strategy='mean/median')
from sklearn.impute import SimpleImputer

Arbitrary value imputation
~~Not used very much~~ handle missing
imputer1 = SimpleImputer(strategy='constant', fill_value=999)

mode
adv → change in distribution, co-relation + addition of outliers
imputer1 = SimpleImputer(strategy='most_frequent')

"missing value imputation"
imputer1 = SimpleImputer(strategy='constant', fill_value='missing')

Random imputation
adv → distribution and variance remain same
div → can not do with sklearn, memory heavy, distribute the covariance

KNN impute → div = slow, memory heavy, adv = accurate
from sklearn.impute import KNNImputer
knn = KNNImputer(n_neighbors=3, weight='distance')
X_train_trf = knn.fit_transform(X_train)
X_test_trf = knn.transform(X_test)

mice from sklearn.impute import IterativeImputer
imputer1 = IterativeImputer(max_iter=1000, random_state=42)

analysis Part

Numerical columns:

variance — `print(x_train['c1'].var())` | same goes for test

~~`print(x_train['c1'].var())`~~

`print(x_train['c1_trf'].var())`

distribution

```
x_train['c1'].plot(kind='kde', ax=ax,
x_train['c1_trf'].plot(kind='kde', ax=ax,
color='green')
```

co-variance — `x_train.cov()`

co-relation — `x_train.corr()`

outliers — `x_train[['c1','c1_trf']].boxplot()`

categorical columns.

change in categories

```
temp = pd.concat(
[
x_train['c1'].value_counts()/len(
x_train['c1'].dropna())

x_train['c1'].value_counts()/len(x_train)
], axis=1)
temp.colums = ['origin', 'imputed']
temp.
```

④ outlier detection

ways to handle outlier → Trimming, Capping, other → treating them as missing, Discretization

ways to detect outlier → zscore treatment, Interquartile proximity rule, percentile based approach

zscore treatment

↳ upper limit ⇒ $\mu + 3\sigma$ < value, lower limit ⇒ $\mu - 3\sigma$ > value

↳ used for normally distributed column

upper_limit = df['c1'].mean() + 3*df['c1'].std()
lower_limit = df['c1'].mean() - 3*df['c1'].std()

Finding outliers

df[(df['c1'] > upper_limit) | (df['c1'] < lower_limit)]

Trimming

new df = df[(df['c1'] < upper_limit) & (df['c1'] > lower_limit)]

Capping

df['c1'] = np.where(df['c1'] > upper_limit, upper_limit, np.where(df['c1'] < lower_limit, lower_limit, df['c1']))

Inter Quantile Proximity Rule (Box Plot)

Q1 = 25 Percentile Q3 = 75 Percentile IQR = Q3 - Q1
upper_limit = Q3 + 1.5 * IQR > value
lower_limit = Q1 - 1.5 * IQR < value

~~Finding outliers~~
~~normal ways~~

Q1 = df['c1'].quantile(0.25)
Q3 = df['c1'].quantile(0.75)
IQR = Q3 - Q1
upper_limit = Q3 + 1.5 * IQR
lower_limit = Q1 - 1.5 * IQR

Finding outliers

(a) ~~above method~~ same as above
(b) sns.boxplot(df['c1'])

Trimming + capping // same as above

Percentile based approach → Here we do not use Box Plot

In this method we assume percentile like max = 99, min = 1

upper_limit = df['c1'].quantile(0.99)
lower_limit = df['c1'].quantile(0.01)

Finding outliers / Trimming / Capping — same

⑤ Feature Scaling

i) Standardization

ii) Normalization → minMaxScaling
→ mean normalization
→ MaxAbsScaling
→ RobustScaling

(i) Standardization
→ outlier remain same

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

(ii) minMaxScaling
→ used when you know min value and max value
→ outlier squeeze, Note → cause change in data

```
from sklearn.preprocessing import MinMaxScaler,
MaxAbsScaler, RobustScaler
```

(iii) mean Normalization → not supported by sklearn,
(iv) RobustScaling → used if data has a lot of outlier
(v) MaxAbsScaling → used if data is sparse (a lot of zeros)

⑥ Encoding of categorical columns

```
No. of unique categories = df['c1'].nunique()
No. of occurence of unique categories = df['c1'].value_counts()
```

Types
- Ordinal categorical data → OrdinalEncoder / LabelEncoder
- Nominal categorical data → OHE

Ordinal encoder - `from sklearn.preprocessing import OrdinalEncoder, LabelEncoder, OneHotEncoder`

```
oe = OrdinalEncoder(categories=[['Poor','Avg','Good'],['D','C','B','A']])
```

Label encoder → `le = LabelEncoder()` // Here we do not mention the categories

OHE →
```
ohe = OneHotEncoder(drop='first', sparse=False, dtype=np.int32)
```

Note → if so many categories then ⇒ threshold = 100,

```
counts = df['c1'].value_counts()
repl = counts[counts <= threshold].index
pd.get_dummies(df['c1'].replace(repl, 'uncommon'))
```

⑦ Encoding of numerical columns

Types → Discretization ↔ custom binning
unsupervised binning → equal width/uniform, equal freq/quantile, kmeans binning
supervised binning
→ Binarization

Equal width → you have to decide number of bins → create equal size bin
(max value − min value) / Number of bins
handle outliers and no change in spread of data
→ Here we to decide bins
Equal freq → handle outliers and normally dist the spread of data
kmean binning → when data is present in clusters

```
from sklearn.preprocessing import KBinsDiscretizer, Binarizer
a = KBinsDiscretizer(n_bins=15, encode='ordinal/onehot/onehot-dense', strategy='uniform/quantile/kmeans')
```

Binarization → 0 or 1 Based on threshold ⇒
a = Binarizer(copy=False, threshold=4.0)

⑧ Transformation of numerical columns

if qq plot = 45° = Normally distributed + kde

types → power → Box cox → not supported by value <= 0, add 0.0001 if min = 0
Box cox → Yeo johnson
→ Functional → log, square root, reciprocal, square

```
from sklearn.preprocessing import PowerTransformer, FunctionTransformer
pt = PowerTransformer() → 'Yeo-Johnson'
pt = PowerTransformer(method='box-cox')
```

log Transform → Right skewed (does not work on −ve data)
~~Reciprocal transform~~ square transform → left skew
~~square root transform~~ ~~does not work on −ve data~~
Best → Yeo johnson

⑨ PCA

```
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
```

T1 = FP
T2 = FN

⑩ column Transformer

```
from sklearn.compose import ColumnTransformer
trf1 = ColumnTransformer(transformers=[
('trf1', ____, [6,8])
('trf2', ____, [3])], remainder='passthrough/drop')
```

⑪ Pipeline

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([('trf1', trf1), ('trf2', trf2) ...])
pipe.fit(X_train, Y_train)
```

binary / multi

macro = nominal / none
weighted = ordinal
precision = type1
recall = type2
f1 = type1+2

Grid Search CV

(cv = cross-validation, int)

```
params = {'trf5__max_depth': [ ... ]}
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(pipe, params, cv=5, scoring='accuracy
/precision/precision_macro/precision_micro/precision_weighted/
recall/recall_macro/recall_micro/recall_weighted/f1/
f1_macro/f1_micro/f1_weighted/roc_auc/roc_auc_ovr/
roc_auc_ovo, roc_auc_ovr_weighted, roc_auc_ovo_weighted)
```

→ classification metrics /neg_mean_absolute_error/ neg_mean_squared_error / neg_root_mean_squared_error/ r2 / explained_variance

grid.best_params_
grid.best_score_

Cross validation

```
from sklearn.model_selection import cross_val_score
cross_val_score(pipe, X_train, Y_train, cv=5, scoring=).mean()
```

Linear Regression

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression(), lr.fit(X_train, Y_train), lr.predict(X_test)
```

polynomial

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=True)
X_train_trans = poly.fit_transform(X_train)
x_test_trans = poly.transform(X_test)
```

→ Apply linear Regression

cross validate → for training and test data
→ condition ← = overfitting
→ then use Regularization

Ridge Regularization (L2)

★ Apply if all columns are imp

```
from sklearn.linear_model import Ridge
```

Lasso Regularization (L1)

★ Apply if all columns are not imp

```
from sklearn.linear_model import Lasso
```

Elastic Net Regularization

★ Apply if you know all colums are imp or not imp

```
from sklearn.linear_model import ElasticNet
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
```

for binary class = lr = LogisticRegression()
for multi class = lr = LogisticRegression(multi_class='over')
Polynomial → same as linear regression

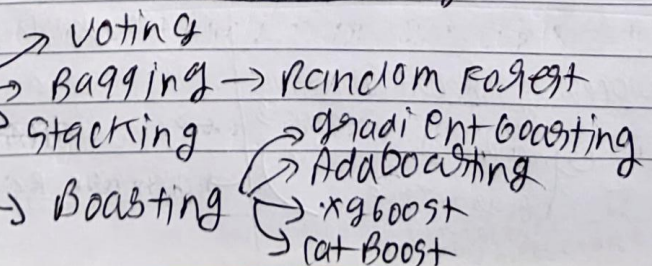
Hyperparameter → penalty = 'l1/l2/elasticnet' → Default = l2
range = [0.01 ..... 100] = C2 = (+int) small value → high Regularization
solver = 5 different default = 'lbfgs'

Decision Tree

```
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
```

Hyperparameters
Regression → criterion = squared_error, poisson, absolute_error, friedman_mse
classifi → criterion = gini, entropy, log_loss
max_depth = 10,
min_samples_split = int default = 2

Ensemble Learning



Voting (use different algo)

Classification → Hard
Regression → Soft

```
from sklearn.ensemble import VotingClassifier, VotingRegressor
lgr = LinearRegression()
dt = DecisionTreeRegressor/Classifier()
svr = SVR()
estimators = [('lgr', lgr), ('dt', dt), ('svr', svr)]
```

Hard / Soft ⇒ `vc = VotingClassifier(estimators=estimators, voting='hard'/'soft')`

⇒ `vc = VotingRegressor(estimators)`

Bagging

(use only 1 algo) DecisionTree

Types → Bagging = row sampling + replacement
→ Pasting = row sampling + not replacement
→ Random subspace = column sampling + with or without replacement
→ Random Patches = column + row sampling + with or without repl

```
from sklearn.ensemble import BaggingClassifier, BaggingRegressor
```

Bagging →
```
bag = BaggingClassifier(
        base_estimator = DTC()
        n_estimators = 500
        max_samples = 0.5  (each model get 0.5 sized d)
        bootstrap = True
        random_state = 42)
```

Pasting →
```
bag = BaggingClassifier(base_estimator=, n_estimators=
max_samples=, bootstrap=False, random_state=42)
```

Random Subspaces →
```
bag = BaggingClassifier(base_estimator,
n_estimators, max_samples=1, bootstrap=False, max_features=0.5,
bootstrap_features=True, random_state=42)
```

Random Patches →
```
bag = BaggingClassifier(base_estimator, n_estimators,
max_samples=0.25, bootstrap=True, max_features=0.5,
bootstrap_features=True, random_state=42)
```

Random Forest

Bagging + DT

```
from sklearn.ensemble import RandomForestClassifier,
RandomForestRegressor
```

Hyper Parameter → n-estimators, max_features, bootstrap = True, max_samples, random_state, oob_score = True + rf.oob_score

AdaBOOST

```
from sklearn.ensemble import AdaBoostClassifier,
AdaBoostRegressor
```

Hyper Parameter → estimator = DecisionTree, n_estimator algorithm = SAMME, SAMME.R, learning_rate = Generally less then 1,

K-mean Clustering

```
from sklearn.cluster import KMeans
wcss []
for i in range(1, 11)
    km = KMeans(n_clusters = i)
    km.fit_predict(df)
    wcss.append(km.inertia_)
plt.plot(range(1, 11), wcss)
```

Gradient Boosting

```
from sklearn.ensemble import GradientBoostingRegressor
GradientBoostingClassifier
```

Hyper Parameter = n_estimator, learning rate = 0.1, max_depth, random_state = 42

Stacking and Blending Ensemble

(use diff Algo)

Here CV = Cross-validation act as K = K-fold mean

```
from sklearn.ensemble import StackingClassifier,
StackingRegressor
```

```
base_model = [
('lr', LinearRegression()),
('rf', RandomForestRegressor(n_estimators=10, random_state=42)),
('gb', GradientBoostingRegressor(n_estimators=10, random_state=42))
]

stack = StackingRegressor(estimators=base_models,
final_estimator = LinearRegression(), cv=10)
```

Hierarchical clustering

```
import scipy.cluster.hierarchy as shc
dend = shc.dendrogram(shc.linkage(data, method='ward'))

from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering(n_clusters=5, affinity=
'euclidean', linkage='ward')
                                    (y_test, y_pred))
```

K Nearest Neighbors

choosing the best model

```
for i in range(1,16):
    knn = KNeighborsClassifier(
        n_neighbors=i)
    knn.fit(x_train, y_train)
    Y_pred = knn.predict(x_test)
    scores.append(accuracy_score
```

```
plt.plot(range(1,16), scores)   // choose the K value
                                   with highest accuracy
```

Now make model → with that K value

SVM

SVM Regressor → not used very much

```
from sklearn.svm import SVC
```

Hyperparameters → C = Regularization parameter, kernel = 'rbf', 'linear'..., degree, random_state = 42

XGBoost

mean 2 class ka hi y

import xgboost as xgb

① xgb_clg = xgb.XGBClassifier(max_depth=6, learning_rate=0.1, n_estimators=120, objective, num_class=2)

② xgb_reg = xgb.XGBRegressor() ← These Hyperparameters

Regression Objective	Classification Objective
'reg: squarederror'	'binary: logistic'
'reg: squaredlogerror'	'binary: logitraw'
'reg: logistic'	'multi: softmax'
'reg: pseudohubererror'	'multi: softprob'

CatBoost

import catboost as cb

① cat_clg = cb.CatBoostClassifier(iterations=100, depth=6, learning_rate=0.1, loss_function='MultiClass')

② cat_reg = cb.CatBoostRegressor()

Regression loss_function	Classification (Logs_function)
RMSE, MAE, Quantile, LogLinQuantile, Poisson	Logloss, CrossEntropy, MultiClass, MultiClassOneVsAll)

LightGBM