IITM MLP

Week 1

- **Pandas**
  - `.iloc` : looks at the integer position
    - `df.iloc[0]` 0th row in the dataframe
    - `df.iloc[4,0]` value at 4th row of the 0th column
  - `.loc` : looks at the index
    - `df.loc[0]` 0th index row of the dataframe
    - `df.loc[4, 'col1']` the value of `col1` at 4-th index
  - conditional slicing
    - `df.loc[df.age < 25]` all the rows where the columns 'age' is less than 25
  - `selector = lambda df: df['col'] > 0`
    - `df.loc[selector]` will give the rows where 'col' values are positive
  - adding a column
    - `df['new_col'] = df['col1'] * 100`
  - modify a value
    - `criteria = df['col'] < 0.2`
    - `df.loc[criteria, 'col'] = 0`
    - `df.loc[df.city == 'Bengalure', ['city' , 'new_city']] = 'Bengaluru'` change the value of the cell where the column 'city' has 'Bengalure' to 'Bengaluru' for the columns 'city' and 'new_city'
  - `df.drop(['col1'], axis=1)` to remove a column
  - `df.drop(['col1'])` to remove a row
  - `df.smaple(n)` n different rows (randomly selected) from the dataframe
    - `df.sample(n, replace=True)` can give the same row multiple times, sampling with replacement
  - `df.groupby('col1').sum()` will give sum of all distinct values in 'col1' by summing the values from other columns

Week 2

1. Look at the big picture

- frame the problem: input/output, business objective
  - supervised, unsupervised or RL?
  - classification/regression?
  - learning style: batch or online?
- select performance measure
  - regression: MSE / MAE
  - classification: precisions, recall, accuracy
- list and check assumptions
  - review assumptions with domain experts

2. Get the data

```
data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/wine-quality-
red.csv'
```

```
data = pd.read_csv(data_url, sep=";")
```

- get familiarised with data by looking at schema and data snippets
- understand the significance of each feature by consulting experts
- use `df.info()` and `df.describe()` to get data type and statistics for each column
- `data['col'].value_counts()` to look at the distribution of values in `col`
- information can be viewed through histogram plots (for each column)
- create a separate test set before exploration in order to have a validation set

  ```
  def split_train_test(data, test_ratio):
          np.random.seed(42)

          shuffled_indices = np.random.permutation(len(data))

          test_set_size = int(len(data) * test_ratio)

          test_indices = shuffled_indices[:test_set_size]
          train_indices = shuffled_indices[test_set_size:]
          return data.iloc[train_indices], data.iloc[test_indices]

  train_set, test_set = split_train_test(data, 0.2)
  ```

  - Scikit-Learn provides functions for creating test sets
    - Random sampling: randomly selects $k\%$ points in the test set
    - Stratified sample: samples test examples such that they are representative of overall distribution

  ```
  from sklearn.model_selection import train_test_split
  train_set, test_set = train_test_split(data, test_size=0.2, random_state=42)

  from sklearn.model_selection import StratifiedShuffleSplit
  split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
  for train_index, test_index in split.split(data, data['label_col']):
          strat_train_set = data.loc[train_index]
          strat_test_set = data.loc[test_index]
  ```

3. Data visualisation

- good idea to create a copy of the training set to avoid any data manipulation in the original set
- scatterplot with seaborn or matplotlib
- correlation matrix between features, use rank correlation for non-linear relationships
- scatter matrix using using `from pandas.plotting`

4. Prepare the date for ML algorithms

- separate features and labels

  ```
  wine_features = strat_train_set.drop("quality",axis=1)
  wine_labels = strat_train_set["quality"].copy()
  ```

- handling missing values/outliers

  ```
  wine_features.isna().sum()
  ```

```
from sklearn.impute import SimpleImputer
# replace missing values with the meadian
imputer = SimpleImputer(strategy="median")
imputer.fit(wine_features)
imputer.statistics_ # gives median values for each column

tr_features = imputer.transform(wine_features)
wine_features_tr = pd.DataFrame(tr_features,columns=wine_features.columns)
```

- feature scaling to bring all features on the same scale

```
# Converting categories into numbers
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
# One Hot encoding (1) if present in that category (0) otherwise
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
```

  - Min-max scaling or Normalisation (0-1 range)
  - Standardisation: subtract mean and divide by standard deviation
- applying transformations like log, square root on the features
- Transformation pipeline:

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
transform_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('std_scaler', StandardScaler())
])
wine_featuers_rt = transform_pipeline.fit_transform(wine_features)
```

- to transform mixed features, where not all features are of the same data type

```
from sklearn.compose import ColumnTransformer
num_attribs = list(wine_features)
cat_attribs = ["place_of_manufacturing"]
full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
])
wine_features_tr = full_pipeline.fit_transform(wine_features)
```

5. Select a model and train it

- good practice to build a quick baseline model on the preprocessed data

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(wine_features_tr, wine_labels)

# Check performance on training set
from sklearn.metrics import mean_squared_error
quality_predictions = lin_reg.predict(wine_features_tr)
mean_squared_error(wine_labels, quality_predictions)

# Check performance on test set
```

```
wine_features_test = strat_test_set.drop("quality", axis=1)
wine_labels_test = strat_test_set["quality"].copy()
wine_features_test_tr = transform_pipeline.fit_transform(wine_features_test)
quality_test_prediction = lin_reg.predict(wine_features_test_tr)
mean_squared_error(wine_labels_test, quality_test_predictions)
```

- DecisionTreeRegressor

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(wine_features, wine_labels)
```

- Cross-validation
    - provides a separate MSE for each validation set, which can use to get and estimation of MSE as well as standard deviation

```
from sklearn.model_selection import cross_val_score
def display_scores(scores):
        print("Scores: ", scores)
        print("Mean: ", scores.mean())
        print("Standard deviation: ", scores.std())

scores = cross_val_score(lin_reg, wine_features_tr,
                                                 wine_labels,
scoring="neg_mean_squared_error", cv=10)
lin_reg_mse_scores = -scores
display_scores(lin_reg_mse_scores)

scores = cross_val_score(tree_reg, wine_features_tr,
                                                 wine_labels,
scoring="neg_mean_squared_error", cv=10)
tree_mse_scores = -scores
display_scores(tree_mse_scores)
```

    - Random forest

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor()
forest_reg.fit(wine_features, wine_labels)

scores = cross_val_score(forest_reg, wine_features_tr, wine_labels,
                                                scoring="neg_mean_squared_error", cv=10)
forest_mse_scores = -scores
display_scores(forest_mse_scores)

quality_test_predictions = forest_reg.predict(wine_features_test_tr)
mean_squared_error(wine_labels_test, quality_test_predictions)
```

6. Fine-tune your model

- Use Grid search for finding the bets combination of hyperparameters
    - RandomForest regression example: Number of estimators, maximum number of features

```
from sklearn.model_selection import GridSearchCV
param_grid = [
        {'n_estimators': [3,10,30], 'max_features': [2,4,6,8]},
        {'bootstrap':[False], 'n_estimators': [3,10], 'max_features': [2,3,4]}
]

grid_search = GridSearchCV(forest_ref, param_grid, cv=5,
                                                        scoring='neg_mean_squared_error',
                                                        return_train_score=True)

grid_search.fit(wine_features_tr, wine_labels)
grid_search.best_params_
grid_search.best_estimator_
```

- Randomised Search: when we have a large hyperparameter space

```
from sklearn.model_selection import RandomizedSearchCV
```

- analysis of best model and its errors

```
feature_importances = grid_search.best_estimator_.feature_importances_
sorted(zip(feature_importances_, feature_list), reverse=True)
```

- evaluation on test set

```
wine_features_test = strat_test_set.drop("quality", axis=1)
wine_labels_test = strat_test["quality"].copy()
wine_features_test_tr = transform_pipeline.fit_transform(wine_features_test)

quality_test_predictions = grid_search.best_estimator_.predict(wine_features_test_tr)
mean_squared_error(wine_labels_test, quality_test_predictions)
```

7. Present your solution

- present solution that highlights learnings, assumptions, and system limitations
- document everything, create clear visualisations and present the model

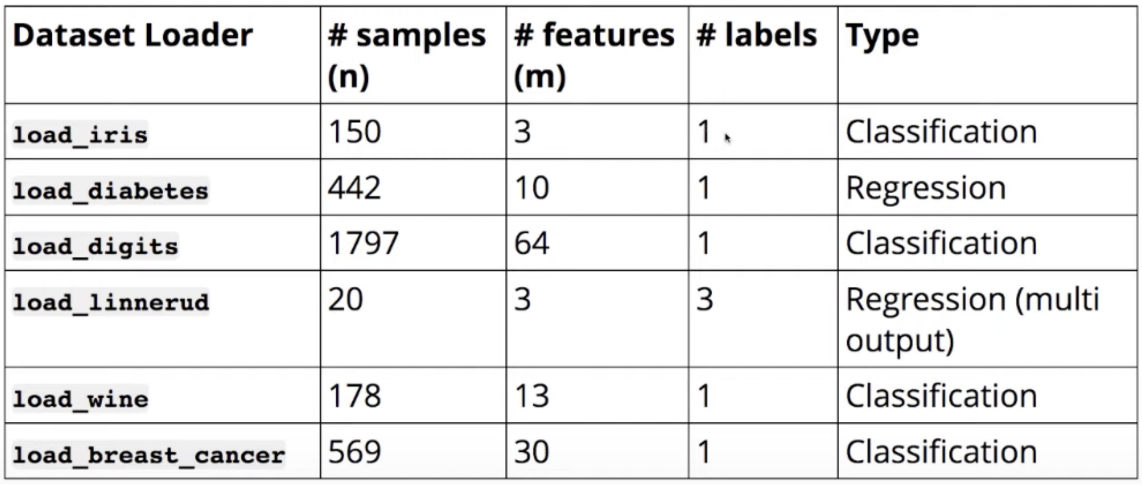
8. Launch, monitor and maintain your system

- Launch
  - plug in input scores
  - write test cases
- Monitoring
  - system outages
  - degradation of model performance
  - sampling predictions for human evaluation
  - regular assessment of data quality; critical for model performance
- Maintenance
  - train model regularly every fixed interval with fresh data
- **SciKit-Learn**
  - sklearn APIs design principles
    - Consistency: All APIs share a simple and consistent interface
    - Inspection : parameters/hyperparameters accessible directly via public instance variables
    - Nonproliferation of classes: datasets are of type Numpy arrays or Scipy sparse matrix
    - Composition: Existing building block are reduced
    - Sensible defaults: values are used for parameters that enable quick baseline building

- Types of sklearn objects
    - Transformers: `transform()` for transforming datasets; `fit()` learns parameters; `fit_transform()` fits parameters and `transform()` the dataset
    - Estimators: `fit()` method
    - Predictions: `predict()` method that takes dataset as an input and returns predictions; `score()` method to measure quality of predictions
- Data API
    - `sklearn.datasets` : loading datasets (custom as well as popular ones)
    - `sklearn.preprocessing` : scaling, centering, normalisation
    - `sklearn.impute` : filling missing values
    - `sklearn.feature_selection` : implements feature selection algorithms
    - `sklearn.feature_extraction` : implements feature extraction from raw data
- Model API
    - implements supervised and unsupervised models
    - Regression
        - `sklearn.linear_model` : linear, ridge, lasso
        - `sklearn.tree`
    - Classification
        - `sklearn.linear_model`
        - `sklearn.svm`
        - `sklearn.trees`
        - `sklearn.neighbours`
        - `sklearn.naive_bayes`
        - `sklearn.multiclass`
    - `sklearn.multioutput` implements multi-output classification and regression
    - `sklearn.cluster` popular clustering algorithms
- Model evaluation API
    - `sklearn.metrics`
    - classification metrics
    - regression metrics
    - clustering metrics
- Model selection API
    - `sklearn.model_selection`
    - cross-validation
    - tuning hyperparameters
    - plotting learning curves
- Model inspection API
    - `sklearn.model_inspection`
    - tools for model inspection

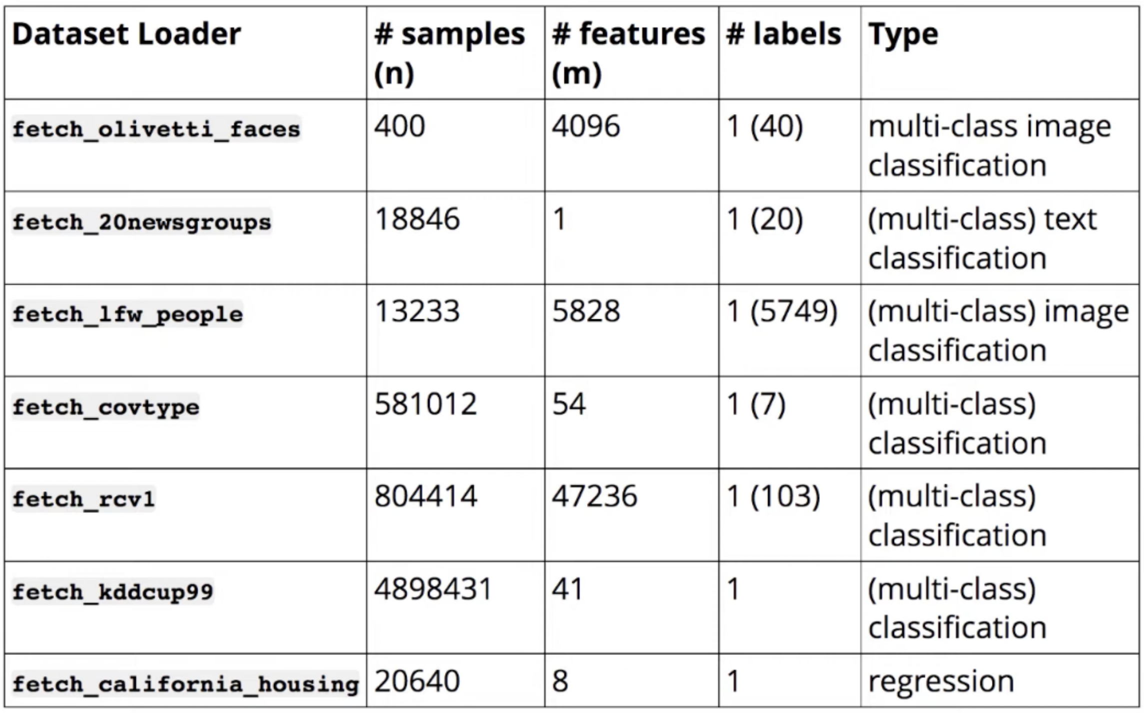
- ## Data Loading
    - Dataset loaders: to load toy dataset bundled with sklearn

    - `load_*`



Dataset Loader	# samples (n)	# features (m)	# labels	Type
`load_iris`	150	3	1	Classification
`load_diabetes`	442	10	1	Regression
`load_digits`	1797	64	1	Classification
`load_linnerud`	20	3	3	Regression (multi output)
`load_wine`	178	13	1	Classification
`load_breast_cancer`	569	30	1	Classification

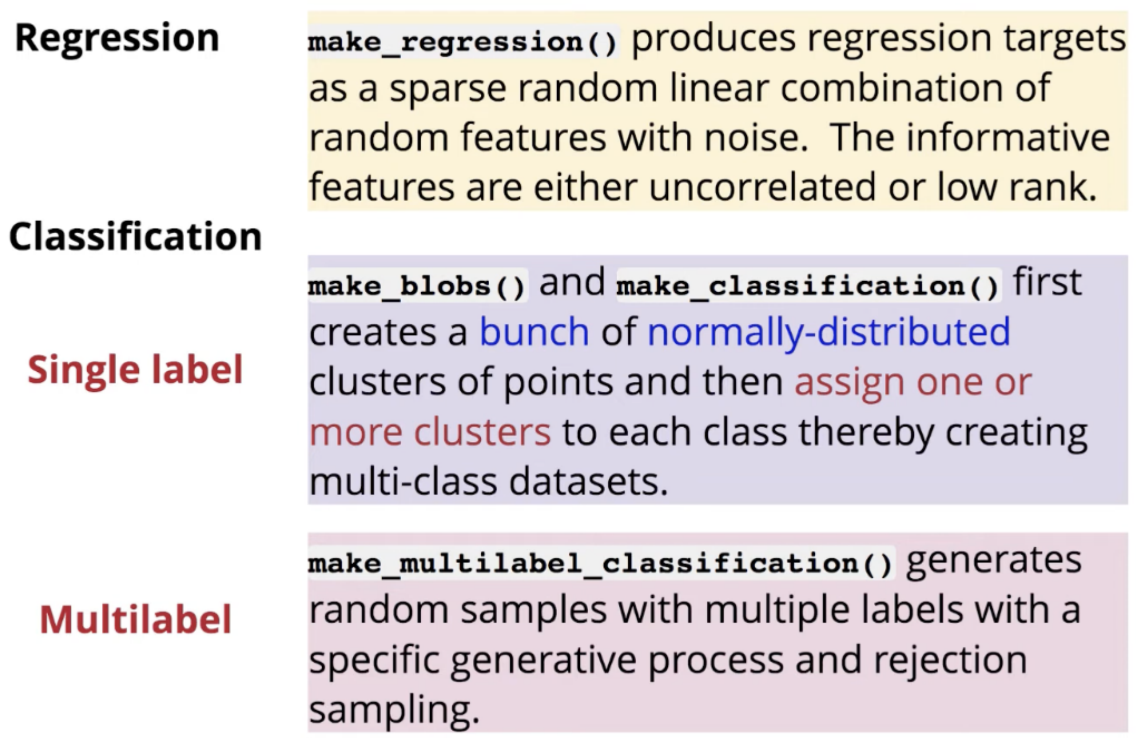
- Dataset fetchers: to download and load datasets from the internet
    - `fetch_*`



Dataset Loader	# samples (n)	# features (m)	# labels	Type
`fetch_olivetti_faces`	400	4096	1 (40)	multi-class image classification
`fetch_20newsgroups`	18846	1	1 (20)	(multi-class) text classification
`fetch_lfw_people`	13233	5828	1 (5749)	(multi-class) image classification
`fetch_covtype`	581012	54	1 (7)	(multi-class) classification
`fetch_rcv1`	804414	47236	1 (103)	(multi-class) classification
`fetch_kddcup99`	4898431	41	1	(multi-class) classification
`fetch_california_housing`	20640	8	1	regression

- Dataset generators: to generate controlled synthetic datasets

- `make_*`



Regression

`make_regression()` produces regression targets as a sparse random linear combination of random features with noise. The informative features are either uncorrelated or low rank.

Classification

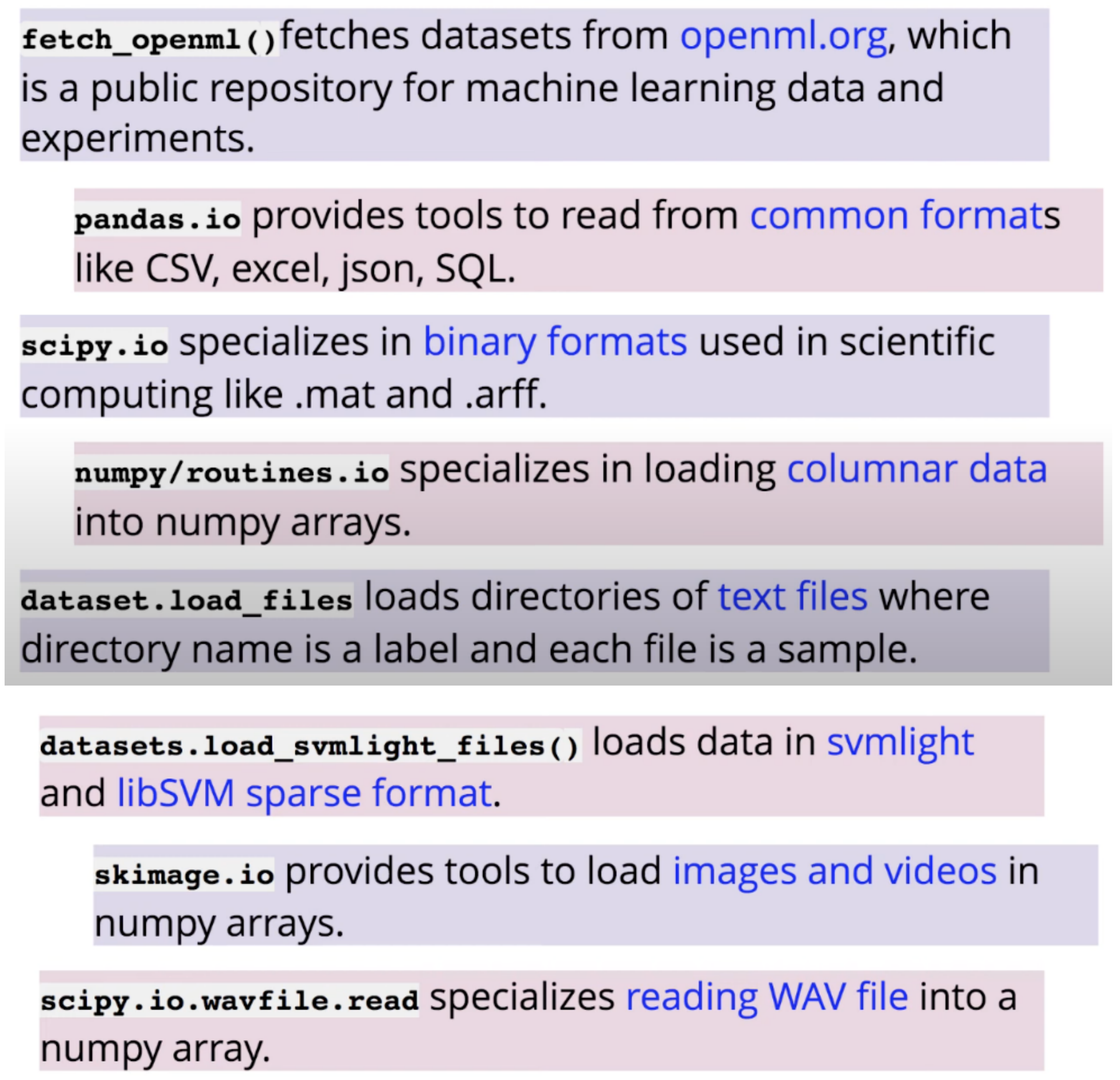
Single label

`make_blobs()` and `make_classification()` first creates a bunch of normally-distributed clusters of points and then assign one or more clusters to each class thereby creating multi-class datasets.

Multilabel

`make_multilabel_classification()` generates random samples with multiple labels with a specific generative process and rejection sampling.

- Loading external datasets



`fetch_openml()` fetches datasets from openml.org, which is a public repository for machine learning data and experiments.

`pandas.io` provides tools to read from common formats like CSV, excel, json, SQL.

`scipy.io` specializes in binary formats used in scientific computing like .mat and .arff.

`numpy/routines.io` specializes in loading columnar data into numpy arrays.

`dataset.load_files` loads directories of text files where directory name is a label and each file is a sample.

`datasets.load_svmlight_files()` loads data in svmlight and libSVM sparse format.

`skimage.io` provides tools to load images and videos in numpy arrays.

`scipy.io.wavfile.read` specializes reading WAV file into a numpy array.

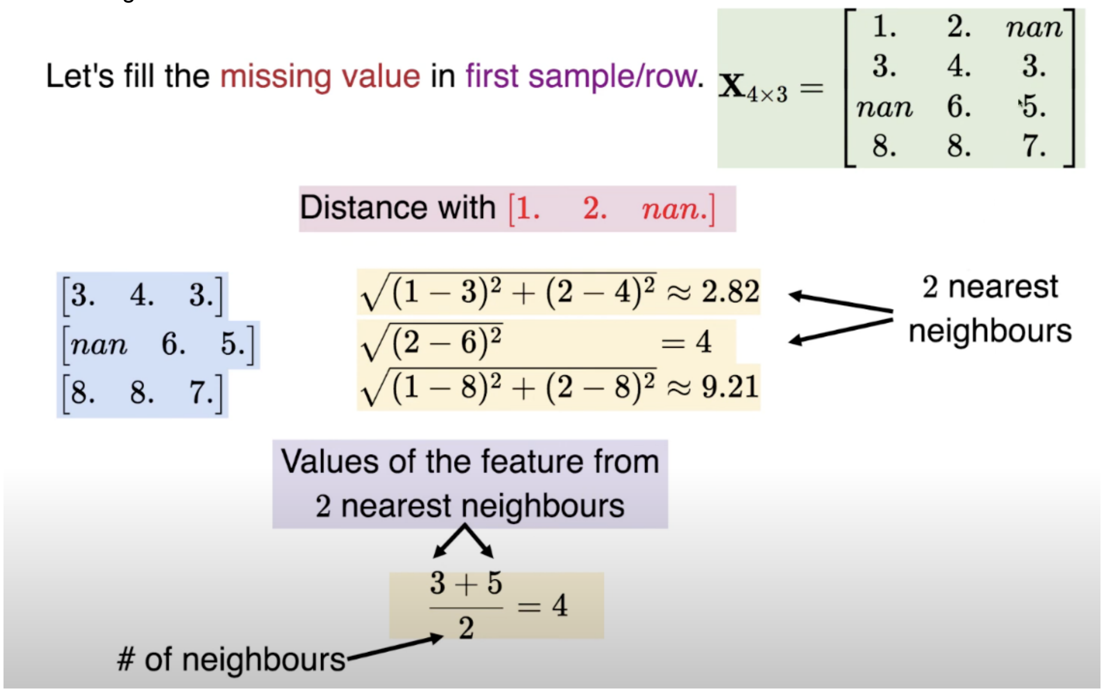
- **sklearn dataset API**

-

Week 3

- **Data Preprocessing**
  - same pre-processing should be applied to both training and test set
  - typical problems:
    - missing values
    - numerical features not on the same scale
    - categorical attributes need to be represented with numerical representation
    - too many features, reduce them
    - extract features from non-numeric data
  - Sklearn library of transformers for preprocessing
    - `sklearn.preprocessing` such as standardisation, missing value imputation, etc.
    - `sklearn.feature_extraction` for feature extraction
    - `sklearn.decomposition.pca` for feature reduction
    - `sklearn.kernel_approximation` for feature expansion
  - Transformer methods
    - `fit()` learns model parameters from a training set
    - `transform()` applies learn transformation to the new data
    - `fit_trandform()` performs both `fit()` and `transform()` and is more efficient
  - **Feature extraction**
    - `sklearn.feature_extraction` has APIs to extract features
    - `DictVectorizer` converts lists of mappings of feature name and feature value, into a matrix (converts a dictionary/dataframe into a matrix)
    - `FeatureHasher` High-speed, low-memory vectoriser that uses feature hashing technique
      - output is `scipy.sparse` matrix
    - `sklearn.feature_extraction.image.*` to extract features from image data
    - `sklearn.feature_extraction.text.*` to extract features from text data
  - **Data Cleaning**
    - `sklearn.impute` to fill missing values in a dataset
    - `SimpleImputer` fills missing values with 'mean', 'median', 'most_frequent' or 'constant'
    - `KNNImputer` fills missing values using *k*-nearest neighbours
      - filled with the mean value of the same attribute of `n_neighbours` closest neighbours

- nearest neighbours are decided based on Euclidean distance


Let's fill the missing value in first sample/row.
$\mathbf{X}_{4\times 3} = \begin{bmatrix} 1. & 2. & nan \\ 3. & 4. & 3. \\ nan & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$
Distance with $[1. \quad 2. \quad nan.]$
$[3. \quad 4. \quad 3.]$
$[nan \quad 6. \quad 5.]$
$[8. \quad 8. \quad 7.]$
$\sqrt{(1-3)^2+(2-4)^2} \approx 2.82$
$\sqrt{(2-6)^2} = 4$
$\sqrt{(1-8)^2+(2-8)^2} \approx 9.21$
2 nearest neighbours
Values of the feature from 2 nearest neighbours
$\frac{3+5}{2} = 4$
# of neighbours

- `MissingIndicator` helps us get the indicators of missing values in the dataset
  - returns a binary matrix, where `True` means missing entry

- **Numeric Transformers**
  - **Feature Scaling**
    - different scales lead to slower convergence; good practice to scale numerical features
    - `StandardScaler`

$$\boldsymbol{x}' = \frac{\boldsymbol{x} - \mu}{\sigma}$$

    - `MaxAbsScaler`

$$\boldsymbol{x}' = \frac{\boldsymbol{x} - \boldsymbol{x}_{\text{min}}}{\boldsymbol{x}_{\text{max}} - \boldsymbol{x}_{\text{min}}}$$

    - `MinMaxScaler`

$$\boldsymbol{x}' = \frac{\boldsymbol{x}}{\text{MaxAbsoluteValue}}$$

    - `FunctionTransformer` applies a user-defined function to transform

```
ft = FunctionTransformer(numpy.log2)
ft.fit_transform(x)
```

  - **Polynomial Transformation**
    - `PolynomialFeatures()` generates a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree

```
pf = PolynomialFeatures(degree=3)
pf.fit_transform(x)
```

  - **Discretization**

        - `KBinsDicretizer` divides a continuous variable into bins; one hot encoding or ordinal encoding is further applied to the bin labels

```
KBinsDiscretizer(
                                n_bins=5,
                                strategy='uniform',
                                encode='ordinal'
)
```

    - **Categorical Transformers**
        - `OneHotEncoder` encodes categorical features/label as one-hot numeric array
            - creates one binary column for each of K unique values

```
ohe = OneHotEncoder()
ohe.fit_transform(x)
```

        - `LabelEncoder` encodes target labels with value between 0 and $K-1$, where K is number of distinct values; can transform only 1-dimensional data
        - `OrdinalEncoder` encodes categorical features with value between 0 and $K-1$; can transform multi-dimensional data
        - `LabelBinarizer`
            - Regression or binary classification can be extended to multi-class setup in one-vs-all fashion
            - Need to convert multi-class labels to binary labels
            - If estimator supports multi-class data, `LabelBinarizer` is not needed
        - `MultiLabelBinarizer`
            - encodes categorical features with value between 0 and $K-1$
        - `add_dummy_feature` augments dataset with a column vector, each value in the column vector is 1
- **Feature Selection**
    - `sklearn.feature_selection` provide many APIs to remove the insignificant features
    - Filter-based methods
        - `VarianceThreshold` removes features with variance below a certain threshold, as specified by the user
        - Univariate scoring functions
            - Mutual information (MI)
                - can be used in classification and regression
                - `mutual_info_regression` and `mutual_info_classif`
                - measures dependency between two variables
                - MI=0 for independent variables
                - higher MI indicates higher dependency
            - F-statistics
                - can be used in classification and regression
                - `f_regression` and `f_classif`
            - Chi-square
                - only for classification problems
                - `chi2`
                - measures dependence b/w two variables
                - computes chi-square stats b/w non-negative feature (bool or frequencies) and class label
                - higher chi-square values indicates that features and labels are likely to be correlated
        - `SelectKBest` removes all but the k highest scoring features
            - To select 20 best features based on chi-square scoring function

```
skb = SelectKBest(chi2, k=20)
X_new = skb.fit_transform(X,y)
```

        - `SelectPercentile` removes all but a user-specified highest scoring percentage of features

            ```
            sp = SelectPercentile(chi2, percentile=20)
            X_new = sp.fit_transform(X,y)
            ```

        - `GenericUnivariateSelect` univariate feature selection with configurable strategy

            ```
            transformer = GenericUnivariateSelect(chi2, mode='k_best', param=20)
            X_new = transformer.fit_transform(X,y)
            ```

    - Wrapper-based methods
        - `RFE` Recursive Feature Elimination
            - recursively removes features using an estimator
            - using *feature importance* from the estimator, removes the least important feature
            - repeat the process until desired number of features are obtained
        - `RFECV`
            - performs RFE in a cross-validation loop to find the optimal number of features
        - `SelectFromModel`
            - selects desired number of important features above certain threshold of feature importance
            - feature importance threshold can be specified numerically or through string argument based on built-in heuristics such 'mean' 'median' and float multiples of like '0.1*mean'

            ```
            clf = LinearSVC(C=0.01, penalty="l1", dual=False)
            clf = clf.fit(X,y)
            clf.coef_

            model = SelectFromModel(clf, prefit=True)
            X_new = model.transform(X)
            ```

        - `SequentialFeatureSelector`
            - selection/deselection features one at a time in a greedy manner
            - ForwardSelection: starts with 0 feature and adds one feature that obtains the best cross-validation score for an estimator when trained on that feature
            - BackwardSelection: starts with all features and removes least important feature by following the idea of forward selection
        - SFS does not require the underlying mode to expose a `coef_` or `feature_importances_` attributes unlike RFE and SelectFromModel
        - SFS may be slower than the other two methods
- **Composite Transformer**
    - `sklearn.compose` is useful to apply transformation on subset of features and combine them
    - `ColumnTransformer`
        - applies a set of transformers to columns of an array or Pandas DataFrame, concatenates the transformed outputs from different transformers into a single matrix
        - useful for transforming heterogenous data by applying transformers to diff subsets of features
        - combines features selection mechanisms and transformation into a single transformer object

        ```
        column_trans = ColumnTransformer(
                                                         [('ageScaler', CountVectorizer(),[0]),
                                                         ('genderEncoder',
        OneHotEncoder(dtype='int'),[1])],
                                                         remainder='drop',
        verbose_feature_names_out=False
        )
        column_trans.fit_transform(X)
        ```

- `TransformedTargetRegressor`
    - transforms target variable
    - takes `regressor` and `transformer` to be applied to target variables as arguments

    ```
    import numpy as np
    from sklearn.linear_model import LinearRegression
    from sklearn.compose import TransformedTargetRegressor
    tt = TransformedTargetRegressor(

    regressor=LinearRegression(),
                                                                    func=np.log,

    inverse_func=np.exp
    )
    X = np.arange(4).reshape(-1,1)
    y = np.exp(2*X).ravel()
    tt.fit(X,y)
    ```

- **Dimensionality Reduction**
    - another way to reduce the number of features is through unsupervised dimensionality reduction techniques
    - `sklearn.decomposition` has APIs for dimensionality reduction
    - PCA uses singular value decomposition (SVD) to project feature matrix or data to a lower dimensional space
    - `sklearn.decomposition.PCA` for PCA based dimensionality reduction
- **Chaining Transformers**
    - example applying multiple transformations

    ```
    si = SimpleImputer()
    X_imputed = si.fit_transform(X)
    ss = StandardScaler()
    X_scaled = ss.fit_transform(X_imputed)
    ```

    - `sklearn.pipeline` to build a composite estimator, as a chain of transformers and estimators
    - `sklearn.pipeline.Pipeline` chain of multiple transformers to execute a fixed sequence of steps
        - intermediate steps of the pipeline must 'transformers' i.e., they must implement fit and transform methods; final estimator only needs to implement fit
        - `Pipeline()` takes a list of tuples, pipeline object exposes interface of the last step

        ```
        estimators = [
                                  ('simpleImputer', SimpleImputer()),
                                  ('standardScaler', StandardScaler()),
        ]
        pipe = Pipeline(steps=estimators)
        pipe.fit_transform(X)
        ```

        - `make_pipeline` takes a number of estimator objects

        ```
        pipe = make_pipeline(
                                              SimpleImputer(),
                                              StandardScaler(),
        )
        ```

        - accessing individual steps in Pipeline

        ```
        estimators = [
                                  ('simpleImputer', SimpleImputer()),
                                  ('pca', PCA()),
                                  ('regressor', LinearRegression())
        ]
        ```

```
pipe = Pipeline(steps=estimators)

# second estimator can be accessed by:
pipe.named_steps.pca
pipe.steps[1]
pipe[1]
pipe['pca']`

# to access parameters of each step in Pipeline
# using syntax <estimator>__<parameterName>
pipe.set_params(pca__n_components = 2)
```

- performing grid search with pipeline

```
param_grid = dict(
                                  imputer=[
                                                      'passthrough',
                                                      SimpleImputer(),
                                                      KNNImputer(),
                                                      clf=[SVC(),
LogisticRegression()],
                                                      clf__C=[0.1, 10, 100]
                                  ]
)
grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

- `sklearn.pipeline.FeatureUnion` combines output from several transformer objects by creating a new transformer from them
  - `FeatureUnion()` accepts a list of tuples

```
num_pipeline = Pipeline([
                                                ('selector', ColumnTransformer([(

'select_first_4',

'passthrough',

slice(0,4))])),
                                                ('imputer',
SimpleImputer(strategy='median')),
                                                ('std_scaler', StandardScaler()),
])
cat_pipeline = ColumnTransformer([
                                                                  ('label_binarizer',
LabelBinarizer(), [4]),
                                                                 ])
full_pipeline = FeatureUnion(
                                          transformer_list = [('num_pipeline',
num_pipeline),

('cat_pipeline', cat_pipeline)]
)
```

- **Demo of Data extraction, imputation, scaling, visualising feature distribution**
  - refer to the `week3.py` file

Week 4

- **Linear Regression**

    - `DummyRegressor` for baseline regression model

      ```
      from sklearn.dummy import DummyRegeressor

      # make prediction based on the mean value
      dummy_regr = DummyRegressor(strategy='mean')
      dummy_regr.fit(X_train, y_train)
      dummy_regr.predict(X_test)
      dummy_regr.score(X_test, y_test)
      ```

        - Strategies: mean, median, quantile, constant
    - Training a Linear Regression model
        - instantiate object (Normal equation or Iterative optimisation)

      ```
      # Normal equation
      from sklearn.linear_model import LinearRegression
      linear_regressor = LinearRegression()

      # Iterative optimisation
      from sklearn.linear_model import SGDRegressor
      linear_regressor = SGDRegressor()
      ```

        - call `fit` on linear regression object `linear_regressor.fit(X_train, y_train)`
- **SGDRegressor Estimator**
    - Stochastic gradient descent
    - use for large training set up (>10k samples)
    - loss functions: 'squared error' or 'huber'
    - penalty: 'l1' or 'l2' or 'elasticnet'
    - learning_rate: 'constant' or 'optimal' or 'invscaling' or 'adaptive'
    - early_stopping: 'True' or 'False'
    - use random seed `linear_regressor = SGDRegressor(random_state=42)`
    - feature scaling for SGD

      ```
      from sklearn.linear_model import SGDRegressor
      from sklearn.pipeline import Pipeline
      from sklearn.preprocessing import StandardScaler

      sgd = Pipeline([
                                      ('feature_scaling', StandardScaler()),
                                      ('sgd_regressor', SGDRegressor())
      ])
      sgd.fit(X_train , y_train)
      ```

    - Learning Rate
        - 'constant' , 'invscaling' , 'adaptive'
        - Default: `learning_rate = 'invscaling'` `eta0 = 1e-2` `power_t = 0.25`
        - learning rate reduces after every iteration: `eta = eta0 / pow(t, power_t)`
        - Constant learning rate
            - `linear_regressor=SGDRegressor(learning_rate='constant', eta0=1e-2)`
        - Adaptive learning rate
            - `linear_regressor=SGDRegressor(learning_rate='adaptive', eta0=1e-2)`
            - learning rate is kept to initial value as long as the training loss decreases
    - set `max_iter` to desired epochs; default is 1000. One epoch is one full pas over the training data
    - Stopping criteria
        - Option 1:

```
linear_regressor = SGDRegressor(
                                                                loss='squared_error',
                                                                max_iter=500,
                                                                tol=1e-3,
                                                                n_iter_no_change=5)
```

        - stops when training loss doesn't improve by more than `tol` for `n_iter_no_change` consecutive epochs
        - else after a maximum number of iteration
    - Option 2:

```
linear_regressor = SGDRegressor(
                                                                loss='squared_error',
                                                                early_stopping=True,
                                                                max_iter=500,
                                                                tol=1e-3,

validation_faction=0.2,
                                                                n_iter_no_change=5

)
```

        - stops when validation score doesn't improve by at least `tol` for `n_iter_no_change` consecutive epochs
  - Averaged SGD
    - `linear_regressor = SGDRegressor(average=True)`
    - `linear_regressor = SGDRegressor(average=10)` starts averaging after 10 samples
    - works best with a large number of features and higher eta0
  - to initialise SGD with weight vector of the previous run, `warm_start=True`

```
sgd_reg = SGDRegressor(max_iter=1, told=-np.infty, warm_start=True,
                                          penalty=None, learning_rate='constant',
eta0=0.0005)

for epoch in range(1000):
        sgd_reg.fit(X_traing, y_train)
        y_val_predict = sgd_reg.predict(X_val)
        val_error = mean_squared_error(y_val, y_val_predict)
```

- **Model Inspection**
  - accessing weights of the model
  - `coef_` class variable stores weights
  - `intercept_` class variable stores intercept
- **Model Inference**
  - Predict labels for feature matrix `X_test`
  - `linear_regressor.predict(X_test)`
- **Model Evaluation**
  - General steps
    1. Split data into train and test
    2. Fit linear regression estimator on training set
    3. Calculate training error (a.k.a. empirical error)
    4. Calculate test error (a.k.a. generalisation error)
  - using `score` method on linear regression object

```
linear_regressor.score(X_test, y_test)
```

  - returns R^2, coefficient of determination

    $$R^2 = (1 - \frac{u}{v})$$

    where

    $$u = (Xw - y)^T (Xw - y)$$

    and

    $$v = (y - \hat{y}_{\text{mean}})^T (y - \hat{y}_{\text{mean}})$$

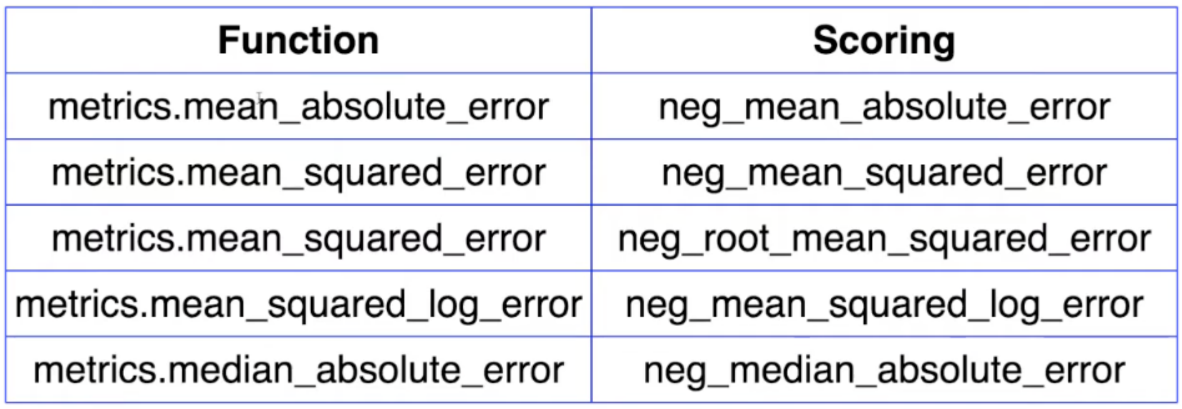
  - best possible score of R^2 is 1.0
  - a constant model that always predicts the expected value of y will get a score of 0 because $u = v$
- Evaluation metrics

  ```
  from sklearn.metrics import mean_absolute_error, mean_squared_error,\
  r2_score, mean_squared_log_error, mean_absolute_percentage_error,\
  mean_absolute_error

  eval_score = mean_absolute_error(y_test, y_predicted)
  eval_score = mean_squared_error(y_test, y_predicted)
  eval_score = r2_score(y_test, y_predicted)

  # for targets with exponential growth like population, sales growth, etc.
  eval_score = mean_squared_log_error(y_test, y_predicted)

  # sensitive to relative error
  eval_score = mean_absolute_percentage_error(y_test, y_predicted)

  # robust to outliers
  eval_score = mean_absolute_error(y_test, y_predicted)
  ```

- worst case error

  ```
  from sklearn.metrics import max_error
  train_error = max_error(y_train, y_predicted)
  test_error = max_error(y_test, y_predicted)
  ```

- score metrics: higher the better; error metrics: lower the better



Function	Scoring
metrics.mean_absolute_error	neg_mean_absolute_error
metrics.mean_squared_error	neg_mean_squared_error
metrics.mean_squared_error	neg_root_mean_squared_error
metrics.mean_squared_log_error	neg_mean_squared_log_error
metrics.median_absolute_error	neg_median_absolute_error

- Cross validation for robust performance evaluation
  - performed by repeated splitting, providing many training and test errors
  - cross validation iterators: `KFold` , `RepeatedKfold` , `LeaveOneOut` , `ShuffleSplit`

    ```
    from sklearn.model_selection import cross_val_score, KFold
    from sklearn.linear_model import linear_regression

    lin_reg = linear_regression()
    ```

```
score = cross_val_score(lin_reg, X, y, cv=5)

# Alternate way of writing the same thing
kfold_cv = KFold(n_splits=5, random_state=42)
score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

- LeaveOneOut

```
from sklearn.model_selection import cross_val_score, LeaveOneOut
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()
loocv = LeaveOneOut()
score = cross_val_score(lin_reg, X, y, cv=loocv)

# Same as using KFold using splits = n
from sklearn.model_selection import KFold
n = X.shape[0]
kfold_cv = KFold(n_splits=n)
score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

- ShuffleSplit

```
from sklearn.model_selection import cross_val_score, ShuffleSplit
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()
shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
score = cross_val_score(lin_reg, X, y, cv=shuffle_split)
```

  - also called random permutation based cross validation strategy
  - user defined number of train/test splits
  - robust to class distribution
- can specify performance measure in `cross_val_score()`

```
cross_val_score(
                                lin_reg,
                                X,
                                y,
                                cv=shufflesplit,
                                scoring='neg_mean_absolute_error'
                                )
```

- to obtain test scores

```
from sklearn.model_selection import cross_validate, ShuffleSplit

cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
cv_results = cross_validate(
                                                        regressor,
                                                        data,
                                                        target,
                                                        cv=cv,

scoring='neg_mean_absolute_error
                                                        )
```

  - results stores in python dictionary with keys:
    - `fit_time` : time required for fitting the model on training set

        - `score_time` : time to get score on evaluation set
        - `test_score` : the actual score obtained on the evaluation set
        - `estimator` : to obtain this, set `return_estimator=True`
        - `train_score` : to obtain this, set `return_train_score=True`
- to study effects of number of samples on train/test errors

  ```
  from sklearn.model_selection import learning_curve

  results = learning_curve(
                                                  lin_reg, X_train, y_train,
                                                  train_sizes=train_sizes, cv = cv,
                                                  scoring='neg_mean_absolute_error'
  )
  train_size, train_scores, test_scores = results[:3]
  # convert scores into errors
  train_errors, test_errors = -train_scores, -test_scores
  ```

-