# Data Preprocessing

Machine Learning for Engineering Applications

Fall 2023

# Dealing with Missing Data

- Missing data from the dataset is very common
- You will see blanks, NULLs, N/A, etc…
- There are techniques to deal with missing data
- Caution:  The wrong technique can cause a bad-design model & accuracies

- Pandas! (Not the cute animal we all love)
- Pandas is a Python library for data analytics
- Pandas is great for importing & managing data in the world of Python
- Pandas is one of the most used libraries for data management in the Machine Learning World!
- Site: https://pandas.pydata.org/

# Importing and identifying missing data

```
import pandas as pd
from io import StringIO

csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''

df = pd.read_csv(StringIO(csv_data))
df
```

| | A | B | C | D |
|---|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 | 4.0 |
| 1 | 5.0 | 6.0 | NaN | 8.0 |
| 2 | 10.0 | 11.0 | 12.0 | NaN |

# Location and count of missing data

```
import pandas as pd
from io import StringIO

csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''

df = pd.read_csv(StringIO(csv_data))
df.isnull().sum*()
```

```
A    0
B    0
C    1
D    1
```

# Numpy Array-like representation of Data Frame

```
import pandas as pd

csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''

df = pd.read_csv(csv_data)
df.values
```

```
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

# Technique of Elimination

## Numpy Array-like representation of Data Frame

```
import pandas as pd

csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''

df = pd.read_csv(csv_data)
df.dropna(axis=0)
```

| | A | B | C | D |
|---|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 | 4.0 |

- axis=0 means:  eliminate rows (samples) with "*nan*" values

## Numpy Array-like representation of Data Frame

```
import pandas as pd

csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''

df = pd.read_csv(csv_data)
df.dropna(axis=1)
```

| | A | B |
|---|---|---|
| 0 | 1.0 | 2.0 |
| 1 | 5.0 | 6.0 |
| 2 | 10.0 | 11.0 |

- axis=1 means: eliminate features (columns) with "*nan*" values

- Threshold operation

```
df.dropna(thresh=4)
```

| | A | B | C | D |
|---|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 | 4.0 |

- Removes samples with less than four feature values in the data frame

- Subset operation

```
df.dropna(subset['C'])
```

| | A | B | C | D |
|---|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 | 4.0 |
| 2 | 10.0 | 11.0 | 12.0 | NaN |

- Goes into feature ‘C’ and remove samples with “*nan*”

- Danger in removing too many samples
  - Analysis is not reliable or impossible
- Danger in removing too many features
  - Valuable feature information missing can affect the label classes needed for the machine to learn from

# Imputing Missing Values

- In statistics, imputation is the process of replacing missing data with substituted values. When substituting for a data point, it is known as "*unit imputation;*" when substituting for a component of a data point, it is known as "*item imputation.*"

  *-- Wikipedia*

- Dr. Valles calls it: **Interpolation**

- There are plenty of interpolation techniques to help in adding/replacing values to your dataset.

# Average/Mean imputation

```python
from sklearn.preprocessing import Imputer

imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
imputed_data
```

By feature

- Median
- Most frequent
- Less frequent
- Etc

| | A | B | C | D |
|---|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 | 4.0 |
| 1 | 5.0 | 6.0 | NaN | 8.0 |
| 2 | 10.0 | 11.0 | 12.0 | NaN |

```
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  7.5,   8.],
       [ 10.,  11.,  12.,   6.]])
```

# Strings & Numerical Values

- Data comes in different **flavors**
- Some values consists of numerical values: integers, decimals, scientific notation, etc.
- **Some values consist of characters**: string of characters, single characters, special characters, etc.
- ML Models evaluate information in a numerical fashion; therefore, we need to “***transform***” string-like values to something that is workable.

- **Ordinal features**: values that can be categorized in a sorted or listed in an understandable order.

- **Example**: Clothing sizes
  - Small < Medium < Large < X-Large < so on
  - These are not numerical
  - However, the order is universally understood

- Ordinal features can then be arranged in such a way the ML model can "*understand*" the order its learning or the impact of such order.

- **Nominal features**: values are understood but do not have an order meaning or impact to its own information.

- **Example**: Colors
  - Blue < Red < Pink < Grey < so on ???
  - These are not numerical, and no order is understood
  - However, the value of the feature is known to be important

- Nominal features will have to be transformed to a numeral form that has represents the same information and it must be impactful to the overall learning of the machine.

```
import pandas as pd

df = pd.DataFrame([
      ... ['green', 'M', 10.1, 'class1'],
      ... ['red', 'L', 13.5, 'class2'],
      ... ['blue', 'XL', 15.3, 'class1']])
df.columns = ['color', 'size', 'price', 'classlabel']
df
```

| | color | size | price | classlabel |
|---|---|---|---|---|
| 0 | green | M | 10.1 | class1 |
| 1 | red | L | 13.5 | class2 |
| 2 | blue | XL | 15.3 | class1 |

Color: Nominal Feature
Size: Ordinal Feature
Price: Numerical Feature

- **Ordinal features**: values can be assigned!

- **Example**: Clothing Sizes

```
size_mapping = {
     ... 'XL': 3,
     ... 'L': 2,
     ... 'M': 1}
df['size'] = df['size'].map(size_mapping)
df
```

| | color | size | price | classlabel |
|---|---|---|---|---|
| 0 | green | 1 | 10.1 | class1 |
| 1 | red | 2 | 13.5 | class2 |
| 2 | blue | 3 | 15.3 | class1 |

# Don’t forget the other ordinal feature!

This is transforming ‘classlabel’ using Numpy

```
import numpy as np

class_mapping = {label:idx for idx,label in
     …enumerate(np.unique(df['classlabel']))}

df['classlabel'] = df['classlabel'].map(class_mapping)

df
```

| | color | size | price | classlabel |
|---|---|---|---|---|
| 0 | green | 1 | 10.1 | 0 |
| 1 | red | 2 | 13.5 | 1 |
| 2 | blue | 3 | 15.3 | 0 |

## Don’t forget the other ordinal feature!...part 2

```
from sklearn.preprocessing import LabelEncoder

class_le = LabelEncoder()

y =
class_le.fit_transform(df['classlabel'].values)

y
```

```
array([0, 1, 0])
```

This is transforming ‘classlabel’ using Scikit-Learn

# Encoding

## (*Nominal*)

- Encoders in Python are great!... Sort of…
- They help to quickly take care of the transformation of nominal features
- The simplest case is the Label Encoder
  - This encoder detects string values
  - Assigns integer values

What’s the problem with this…?

# Example:

```
X = df[['color', 'size', 'price']].values

color_le = LabelEncoder()

X[:, 0] = color_le.fit_transform(X[:, 0])

X
```

blue = 0
green = 1
red = 2

```
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

**One-Hot Encoding**: This will help with the problem below

- This technique will help create temporary (dummy) features to better break the Nominal feature.
- Binary representations for each feature
- Expand the number of columns to add all possible differences

- **Example**

```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X)
```

COLOR

```
array([[  0. ,    1. ,    0. ,    1. ,   10.1],
       [  0. ,    0. ,    1. ,    2. ,   13.5],
       [  1. ,    0. ,    0. ,    3. ,   15.3]])
```

# Don’t Forget… Partition your Data

```python
from sklearn.model_selection import train_test_split

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

X_train, X_test, y_train, y_test =\
     ... train_test_split(X, y,
     ... test_size=0.3,
     ... random_state=0,
     ... stratify=y)
```

# Normalization
# &
# Standardization

## Already covered this one

- Standardized the feature range where the Mean = 0 and Standard Deviation = 1

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

```
from sklearn.preprocessing import
StandardScaler

stdsc = StandardScaler()

X_train_std = stdsc.fit_transform(X_train)

X_test_std = stdsc.transform(X_test)
```

| Input | Standardized |
|---|---|
| 0.0 | -1.46385 |
| 1.0 | -0.87831 |
| 2.0 | -0.29277 |
| 3.0 | 0.29277 |
| 4.0 | 0.87831 |
| 5.0 | 1.46385 |

**Normalization** transforms the range of the feature [0,1]

- Knowing the minimum & maximum is crucial
- Therefore, Python has the magical min-max scaling capabilities

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()

X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

- For every data sample in the feature:

- Not all models require features to be normalized or standardize

- Random Forest, Decision Tree, *k*-NNs do not really benefit from having its features scaled.

- Some highly require it due to the design and architecture of the model (*perceptron*)

- Research will lead you to the right answers!

# Impactful Features

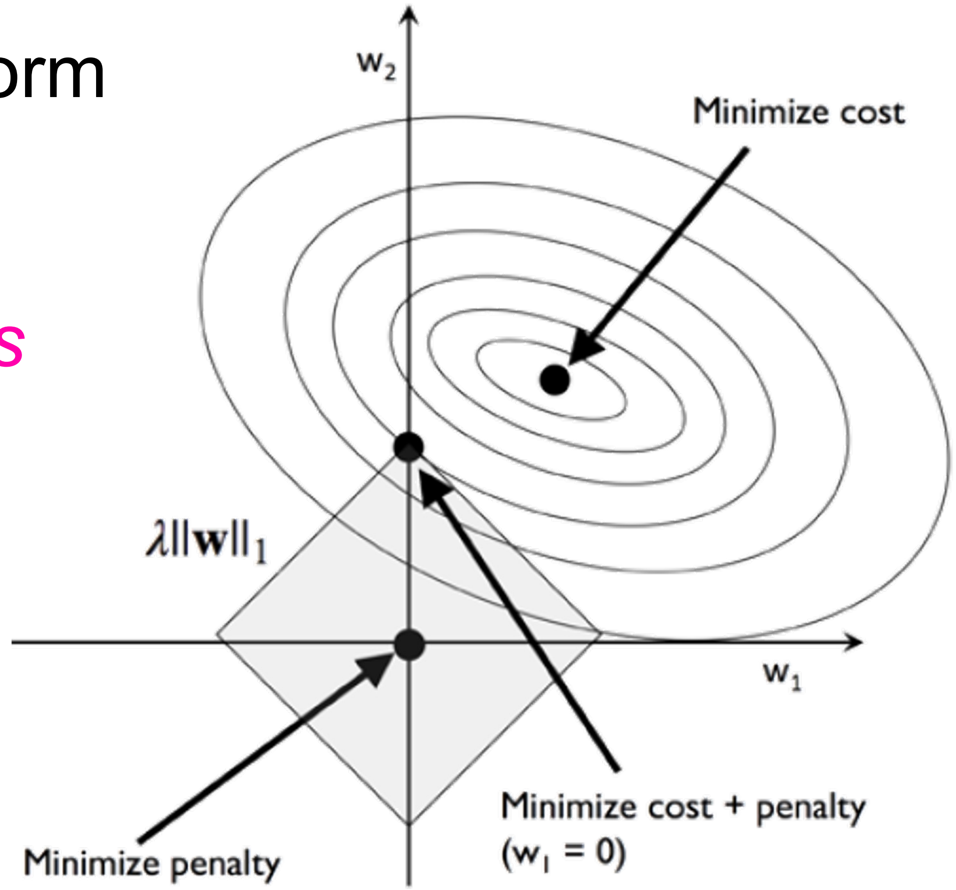
- **L1** yields to spares feature vectors (many zeros for feature weights)

- **L1** is mainly used as the technique to perform feature selection

- This is useful for *high dimensional datasets*

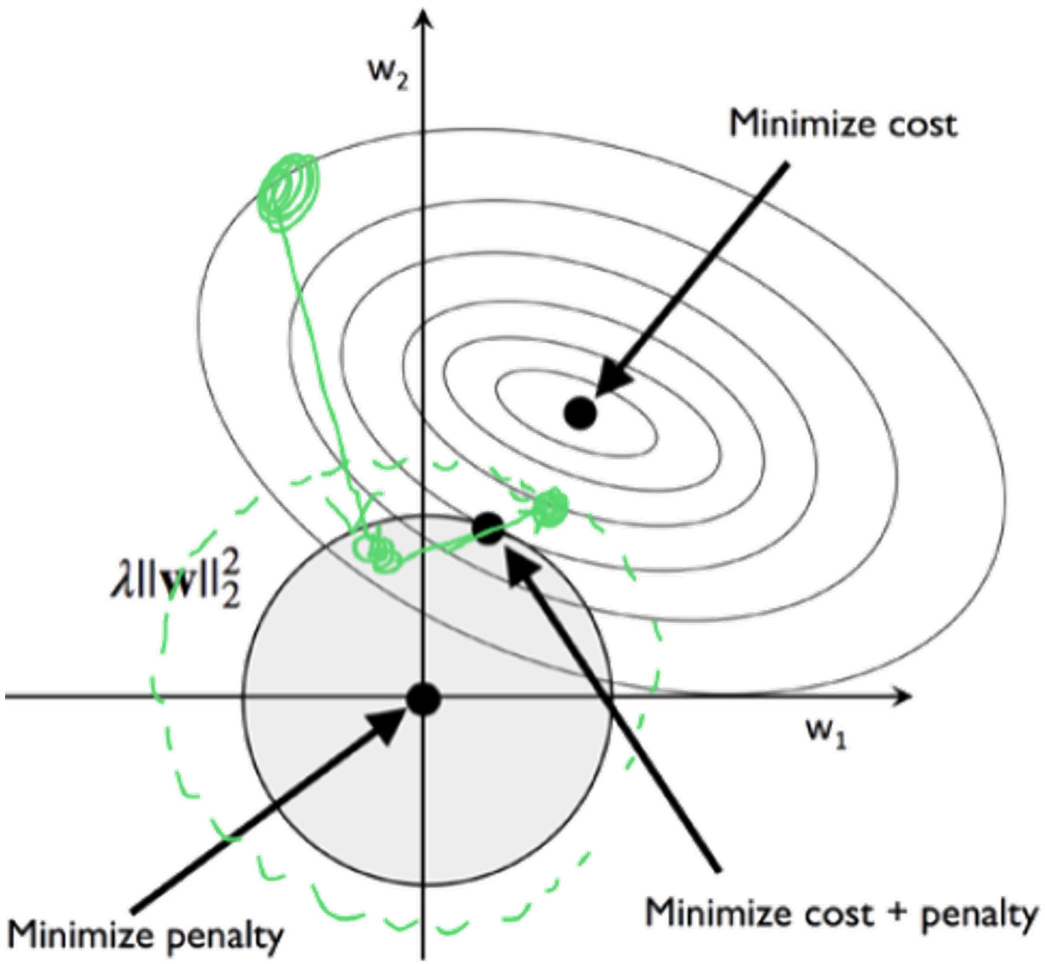
- The **L1 Penalty**:

$$L1: \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$



- **L2** is a penalty term that is added to the cost function of the model
- **L2** help to regulate the size of the big weights vs smaller weights
- The penalty strength is the *lambda* term ($\lambda$)
- The **L2 Penalty**:

$$L2: \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$



```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='l1', C=1.0)
lr.fit(X_train_std, y_train)
print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 1.0
print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 1.0

lr.coef_
array([[1.24559337, 0.18041967, 0.74328894, -1.16046277, 0., 0., 1.1678711,
0., 0., 0., 0., 0.54941931, 2.51017406],

[-1.53720749, -0.38727002, -0.99539203, 0.3651479, -0.0596352 , 0.,
0.66833149, 0., 0., -1.9346134, 1.23297955, 0., -2.23135027],

[ 0.13579227, 0.16837686, 0.35723831, 0., 0., 0., -2.43809275, 0., 0.,
1.56391408, -0.81933286, -0.49187817, 0.]])
```

# Feature Selection & Extraction

- One of the ways to avoid overfitting: **Dimensionality reduction**

- **Two types**:
  - Feature Selection
  - Feature Extraction

- **Feature Selection**: generate a subset of features

- **Feature Extraction**: create a new feature from features

# Next Assignment