GLS University
FCAIT
IMCA SEM VI
ML Notes
Unit 1 Machine learning Practicals

One hot encoding (OHE)

One hot encoding (OHE) is a machine learning technique that encodes categorical data to numerical ones. If you want to perform one hot encoding, both sklearn. preprocessing. OneHotEncoder and pandas. get_dummies are popular choices.

pandas.get_dummies

pandas.get_dummies(*data*, *prefix=None*, *prefix_sep='_'*, *dummy_na=False*, *columns=None*, *sparse=False*, *drop_first=False*, *dtype=None*)[source]

Convert categorical variable into dummy/indicator variables.

Each variable is converted in as many 0/1 variables as there are different values. Columns in the output are each named after a value; if the input is a DataFrame, the name of the original variable is prepended to the value.

Parameters:

dataarray-like, Series, or DataFrame

Data of which to get dummy indicators.

prefixstr, list of str, or dict of str, default None

String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

prefix_sepstr, default '_'

If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

dummy_nabool, default False

Add a column to indicate NaNs, if False NaNs are ignored.

columnslist-like, default None

Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object*, *string*, or *category* dtype will be converted.

sparsebool, default False

Whether the dummy-encoded columns should be backed by a `SparseArray` (True) or a regular NumPy array (False).

drop_firstbool, default False

Whether to get k-1 dummies out of k categorical levels by removing the first level.

dtypedtype, default bool

Data type for new columns. Only a single dtype is allowed.

Returns:

DataFrame

Dummy-coded data. If *data* contains other columns than the dummy-coded one(s), these will be prepended, unaltered, to the result.

OneHotEncoder

class sklearn.preprocessing.OneHotEncoder(*, *categories='auto'*, *drop=None*, *sparse_output=True*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*, *min_frequency=None*, *max_categories=None*, *feature_name_combiner='concat'*)[source]

Encode categorical features as a one-hot numeric array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are encoded using a one-hot (aka ‘one-of-K’ or ‘dummy’) encoding scheme. This creates a binary column for each category and returns a sparse matrix or dense array (depending on the `sparse_output` parameter).

By default, the encoder derives the categories based on the unique values in each feature. Alternatively, you can also specify the `categories` manually.

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

Note: a one-hot encoding of y labels should use a LabelBinarizer instead.

Read more in the User Guide. For a comparison of different encoders, refer to: Comparing Target Encoder with Other Encoders.

Parameters:

categories‘auto’ or a list of array-like, default=’auto’

Categories (unique values) per feature:

- ‘auto’ : Determine categories automatically from the training data.
- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values within a single feature, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

Added in version 0.20.

drop{‘first’, ‘if_binary’} or an array-like of shape (n_features,), default=None

Specifies a methodology to use to drop one of the categories per feature. This is useful in situations where perfectly collinear features cause problems, such as when feeding the resulting data into an unregularized linear regression model.

However, dropping one category breaks the symmetry of the original representation and can therefore induce a bias in downstream models, for instance for penalized linear classification or regression models.

- None : retain all features (the default).
- ‘first’ : drop the first category in each feature. If only one category is present, the feature will be dropped entirely.
- ‘if_binary’ : drop the first category in each feature with two categories. Features with 1 or more than 2 categories are left intact.
- array : `drop[i]` is the category in feature `X[:, i]` that should be dropped.

When `max_categories` or `min_frequency` is configured to group infrequent categories, the dropping behavior is handled after the grouping.

Added in version 0.21: The parameter `drop` was added in 0.21.

Changed in version 0.23: The option `drop='if_binary'` was added in 0.23.

Changed in version 1.1: Support for dropping infrequent categories.

sparse_outputbool, default=True

When `True`, it returns a `scipy.sparse.csr_matrix`, i.e. a sparse matrix in “Compressed Sparse Row” (CSR) format.

Added in version 1.2: `sparse` was renamed to `sparse_output`

dtypenumber type, default=np.float64

Desired dtype of output.

handle_unknown{‘error’, ‘ignore’, ‘infrequent_if_exist’, ‘warn’}, default=’error’

Specifies the way unknown categories are handled during `transform`.

- ‘error’ : Raise an error if an unknown category is present during transform.
- ‘ignore’ : When an unknown category is encountered during transform, the resulting one-hot encoded columns for this feature will be all zeros. In the inverse transform, an unknown category will be denoted as None.

- ‘infrequent_if_exist’ : When an unknown category is encountered during transform, the resulting one-hot encoded columns for this feature will map to the infrequent category if it exists. The infrequent category will be mapped to the last position in the encoding. During inverse transform, an unknown category will be mapped to the category denoted `'infrequent'` if it exists. If the `'infrequent'` category does not exist, then `transform` and `inverse_transform` will handle an unknown category as with `handle_unknown='ignore'`. Infrequent categories exist based on `min_frequency` and `max_categories`. Read more in the User Guide.

- ‘warn’ : When an unknown category is encountered during transform a warning is issued, and the encoding then proceeds as described for `handle_unknown="infrequent_if_exist"`.

Changed in version 1.1: `'infrequent_if_exist'` was added to automatically handle unknown categories and infrequent categories.

Added in version 1.6: The option `"warn"` was added in 1.6.

min_frequencyint or float, default=None

Specifies the minimum frequency below which a category will be considered infrequent.

- If `int`, categories with a smaller cardinality will be considered infrequent.

- If `float`, categories with a smaller cardinality than `min_frequency * n_samples` will be considered infrequent.

Added in version 1.1: Read more in the User Guide.

max_categoriesint, default=None

Specifies an upper limit to the number of output features for each input feature when considering infrequent categories. If there are infrequent categories, `max_categories` includes the category representing the infrequent categories along with the frequent categories. If `None`, there is no limit to the number of output features.

Added in version 1.1: Read more in the User Guide.

feature_name_combiner“concat” or callable, default=”concat”

Callable with signature `def callable(input_feature, category)` that returns a string. This is used to create feature names to be returned by `get_feature_names_out`.

`"concat"` concatenates encoded feature name and category with `feature + "_" + str(category)`.E.g. feature X with values 1, 6, 7 create feature names `X_1, X_6, X_7`.

Added in version 1.3.

Attributes:

categories_list of arrays

The categories of each feature determined during fitting (in order of the features in X and corresponding with the output of `transform`). This includes the category specified in `drop` (if any).

drop_idx_array of shape (n_features,)

- `drop_idx_[i]` is the index in `categories_[i]` of the category to be dropped for each feature.
- `drop_idx_[i] = None` if no category is to be dropped from the feature with index `i`, e.g. when `drop='if_binary'` and the feature isn't binary.
- `drop_idx_ = None` if all the transformed features will be retained.

If infrequent categories are enabled by setting `min_frequency` or `max_categories` to a non-default value and `drop_idx[i]` corresponds to a infrequent category, then the entire infrequent category is dropped.

Changed in version 0.23: Added the possibility to contain `None` values.

`infrequent_categories_`list of ndarray

Infrequent categories for each feature.

n_features_in_int

Number of features seen during fit.

MinMaxScaler

There is another way of data scaling, where the minimum of feature is made equal to zero and the maximum of feature equal to one. MinMax Scaler shrinks the data within the given range, usually of 0 to 1. It transforms data by scaling features to a given range. It scales the values to a specific value range without changing the shape of the original distribution.

class sklearn.preprocessing.MinMaxScaler(*feature_range=(0, 1), *, copy=True, clip=False*) [source]

Transform features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

This transformation is often used as an alternative to zero mean, unit variance scaling.

`MinMaxScaler` doesn't reduce the effect of outliers, but it linearly scales them down into a fixed range, where the largest occurring data point corresponds to the maximum value and the smallest one corresponds to the minimum value. For an example visualization, refer to Compare MinMaxScaler with other scalers.

Read more in the User Guide.

Parameters:

feature_rangetuple (min, max), default=(0, 1)

Desired range of transformed data.

copybool, default=True

Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

clipbool, default=False

Set to True to clip transformed values of held-out data to provided `feature range`.

Added in version 0.24.

Attributes:

min_ndarray of shape (n_features,)

Per feature adjustment for minimum. Equivalent to `min - X.min(axis=0) * self.scale_`

scale_ndarray of shape (n_features,)

Per feature relative scaling of the data. Equivalent to `(max - min) / (X.max(axis=0) - X.min(axis=0))`

Added in version 0.17: *scale_* attribute.

data_min_ndarray of shape (n_features,)

Per feature minimum seen in the data

Added in version 0.17: *data_min_*

data_max_ndarray of shape (n_features,)

Per feature maximum seen in the data

Added in version 0.17: *data_max_*

data_range_ndarray of shape (n_features,)

Per feature range `(data_max_ - data_min_)` seen in the data

Added in version 0.17: *data_range_*

n_features_in_int

Number of features seen during fit.

Added in version 0.24.

n_samples_seen_int

The number of samples processed by the estimator. It will be reset on new calls to fit, but increments across `partial_fit` calls.

feature_names_in_ndarray of shape (`n_features_in_`,)

Names of features seen during fit. Defined only when `X` has feature names that are all strings.

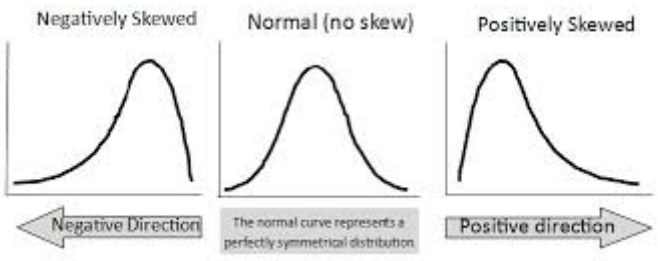
Introduction:

In machine learning, data comes in all shapes and sizes. However, not all data is created equal. One common challenge that data scientists often encounter is skewness. Skewed data can adversely affect the performance of machine learning models, leading to inaccurate predictions and biased results. This blog post will delve into skewness, explore its implications in machine learning, and discuss various techniques to handle and transform skewed data effectively.

Understanding Skewness

Skewness refers to the asymmetry or lack of symmetry in the distribution of data. A dataset is considered skewed if the distribution is not symmetrical, meaning that the tail of the distribution is longer on one side than the other. Skewness can manifest in two forms:

1. **Positive Skewness (Right-skewed)**: In a positively skewed distribution, the tail of the distribution extends to the right, indicating that the majority of the data points are concentrated on the left side, with a few outliers on the right.
2. **Negative Skewness (Left-skewed):** Conversely, in a negatively skewed distribution, the tail of the distribution extends to the left, indicating that the majority of the data points are concentrated on the right side, with a few outliers on the left.



Implications of Skewed Data in Machine Learning: Skewed data can pose several challenges in the context of machine learning:

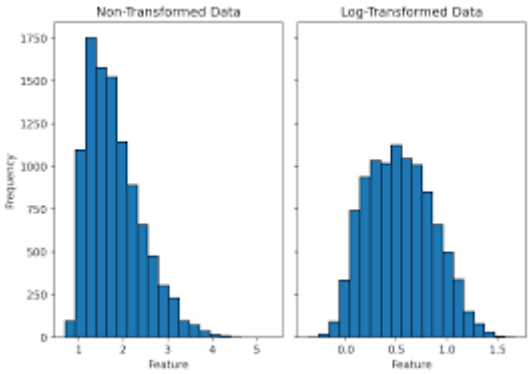
1. **Biased Models**: Machine learning models trained on skewed data may exhibit bias towards the majority class or the dominant range of values, leading to poor generalization performance on unseen data.
2. **Inaccurate Predictions**: Skewed data can distort the relationship between input features and target variables, resulting in inaccurate predictions and unreliable insights.
3. **Suboptimal Performance:** Skewed data can hinder the performance of various machine learning algorithms, especially those that assume a normal distribution of data, such as linear regression and logistic regression.

Handling Skewness:

Techniques and Transformations To mitigate the effects of skewness and improve the performance of machine learning models, data preprocessing techniques and transformations can be applied. Here are some common approaches:

Log Transformation:

- Log transformation is particularly effective for reducing right-skewness in data.
- It involves applying the natural logarithm (or other logarithmic functions) to the skewed feature, which compresses large values and expands small values.
- Log transformation can help stabilize variance and make the distribution more symmetrical.



Tokenization

Tokenization is a crucial step in NLP text preprocessing where text is segmented into smaller units, typically words or subwords, known as tokens. This process is essential for several reasons. Firstly,

it breaks down the text into manageable units for analysis and processing. Secondly, it standardizes the representation of words, enabling consistency in language modeling tasks. Additionally, tokenization forms the basis for feature extraction and modeling in NLP, facilitating tasks such as sentiment analysis, named entity recognition, and machine translation. Overall, tokenization plays a fundamental role in preparing text data for further analysis and modeling in NLP applications.

We Generally do 2 Type of tokenization 1. Word tokenization 2. Sentence Tokenization

9.1 NLTK

NLTK is a Library used to tokenize text into sentences and words.

```
# Import Libraray
from nltk.tokenize import word_tokenize,sent_tokenize
```

What is the Natural Language Toolkit (NLTK)?

As discussed earlier, NLTK is Python's API library for performing an array of tasks in human language. It can perform a variety of operations on textual data, such as classification, tokenization, stemming, tagging, Leparsing, semantic reasoning, etc.

Installation:
NLTK can be installed simply using pip or by running the following code.

```
! pip install nltk
```

Accessing Additional Resources:
To incorporate the usage of additional resources, such as recourses of languages other than English – you can run the following in a python script. It has to be done only once when you are running it for the first time in your system.

```
import nltk
nltk.download('all')
```

Now, having installed NLTK successfully in our system, let's perform some basic operations on text data using NLTK.

Tokenization

Tokenization refers to break down the text into smaller units. It entails splitting paragraphs into sentences and sentences into words. It is one of the initial steps of any NLP pipeline. Let us have a look at the two major kinds of tokenization that NLTK provides:

Work Tokenization

It involves breaking down the text into words.

```
 "I study Machine Learning on GeeksforGeeks." will be word-
tokenized as
  ['I', 'study', 'Machine', 'Learning', 'on', 'GeeksforGeeks',
'.'].
```

Sentence Tokenization

It involves breaking down the text into individual sentences.

Example:

```
"I study Machine Learning on GeeksforGeeks. Currently, I'm
studying NLP"
 will be sentence-tokenized as
 ['I study Machine Learning on GeeksforGeeks.', 'Currently, I'm
studying NLP.']
```

In Python, both these tokenizations can be implemented in NLTK as follows:

```
# Tokenization using NLTK
from nltk import word_tokenize, sent_tokenize
sent = "GeeksforGeeks is a great learning platform.\
It is one of the best for Computer Science students."
print(word_tokenize(sent))
print(sent_tokenize(sent))
```

Output:

```
['GeeksforGeeks', 'is', 'a', 'great', 'learning', 'platform', '.',
'It', 'is', 'one', 'of', 'the', 'best', 'for', 'Computer',
'Science', 'students', '.']
['GeeksforGeeks is a great learning platform.',
 'It is one of the best for Computer Science students.']
```

Stemming and Lemmatization

When working with Natural Language, we are not much interested in the form of words – rather, we are concerned with the meaning that the words intend to convey. Thus, we try to map every word of the language to its root/base form. This process is called canonicalization.

E.g. The words ‘play’, ‘plays’, ‘played’, and ‘playing’ convey the same action – hence, we can map them all to their base form i.e. ‘play’.

Now, there are two widely used canonicalization techniques: **Stemming** and **Lemmatization.**

Stemming

Stemming generates the base word from the inflected word by removing the affixes of the word. It has a set of pre-defined rules that govern the dropping of these affixes. It must be noted that stemmers might not always result in semantically meaningful base words. Stemmers are faster and computationally less expensive than lemmatizers.

In the following code, we will be stemming words using Porter Stemmer – one of the most widely used stemmers:

```
from nltk.stem import PorterStemmer

# create an object of class PorterStemmer
porter = PorterStemmer()
print(porter.stem("play"))
print(porter.stem("playing"))
```

```
print(porter.stem("plays"))
print(porter.stem("played"))
```

Output:

```
play
play
play
play
```

We can see that all the variations of the word ‘play’ have been reduced to the same word – ‘play’. In this case, the output is a meaningful word, ‘play’. However, this is not always the case. Let us take an example.

Please note that these groups are stored in the lemmatizer; there is no removal of affixes as in the case of a stemmer.

```
from nltk.stem import PorterStemmer
# create an object of class PorterStemmer
porter = PorterStemmer()
print(porter.stem("Communication"))
```

Output:

```
commun
```

The stemmer reduces the word ‘communication’ to a base word ‘commun’ which is meaningless in itself.

Lemmatization

Lemmatization involves grouping together the inflected forms of the same word. This way, we can reach out to the base form of any word which will be meaningful in nature. The base from here is called the Lemma.

Lemmatizers are slower and computationally more expensive than stemmers.

Example:
'play', 'plays', 'played', and 'playing' have 'play' as the lemma.

In Python, both these tokenizations can be implemented in NLTK as follows:

```
from nltk.stem import WordNetLemmatizer
# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("plays", 'v'))
print(lemmatizer.lemmatize("played", 'v'))
print(lemmatizer.lemmatize("play", 'v'))
print(lemmatizer.lemmatize("playing", 'v'))
```

Output:

```
play
play
play
```

```
play
```

Please note that in lemmatizers, we need to pass the Part of Speech of the word along with the word as a function argument.

Also, lemmatizers always result in meaningful base words. Let us take the same example as we took in the case for stemmers.

```
from nltk.stem import WordNetLemmatizer

# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("Communication", 'v'))
```

Output:

```
Communication
```

Stemming and lemmatization are two text preprocessing techniques used to reduce words to their base or root form. The primary goal of these techniques is to reduce the number of unique words in a text document, making it easier to analyze and understand.

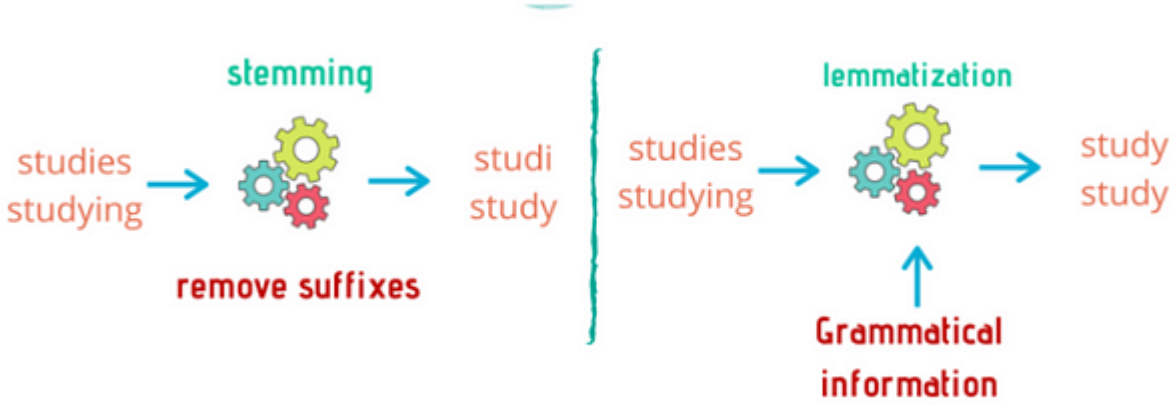
They are widely used for Search engines and tagging. Search engines use stemming for indexing the words. Therefore, instead of storing all forms of a word, a search engine may only store its roots. In this way, stemming reduces the size of the index and increases retrieval accuracy.

Let’s learn them deeply!

Stemming involves removing suffixes from words to obtain their base form while lemmatization involves converting words to their morphological base form.

Stemming is a **simpler** and **faster** technique compared to lemmatization. It uses a set of rules or algorithms to remove suffixes and obtain the base form of a word. However, stemming can sometimes produce a base form that is not valid, in which case it can also lead to ambiguity.

On the other hand, lemmatization is a more sophisticated technique that uses vocabulary and morphological analysis to determine the base form of a word. Lemmatization is **slower** and **more complex** than stemming. It produces a valid base form that can be found in a dictionary, making it more accurate than stemming.



Stemming is preferred when the meaning of the word is not important for analysis. for example: Spam Detection

Lemmatization would be recommended when the meaning of the word is important for analysis. for example: Question Answer

Lemmatization would be recommended when the meaning of the word is **important** for analysis. for example: **Question Answer**

```
Porter stemming algorithm is one of the most common stemming
algorithms which is basically designed to remove and replace well-
known suffixes of English words.
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()

def stem_words(text):
    word_tokens = text.split()
    stems = [stemmer.stem(word) for word in word_tokens]
    return stems

text = 'text preprocessing techniques for natural language
processing by rinkal  shah'
stem_words(text)
```

Output:

```
['text',
 'preprocess',
 'techniqu',
 'for',
 'natur',
 'languag',
 'process',
 'by',
 'rinkal',
 'shah']
```

Now let’s consider the topic of “Lemmatization”

In our lemmatization example, we will be using a popular lemmatizer called **WordNet** lemmatizer.

WordNet is a word association database for English and a useful resource for English lemmatization.

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
def lemmatize_word(text):
    word_tokens = text.split()
    lemmas = [lemmatizer.lemmatize(word, pos ='v') for word in
word_tokens]
    return lemmas

text = 'text preprocessing techniques for natural language
processing by rinkal shah’
lemmatize_word(text)
```

Output:

```
['text',
 'preprocessing',
 'techniques',
 'for',
 'natural',
 'language',
 'process',
 'by',
 'rinkal',
 'shah']
```