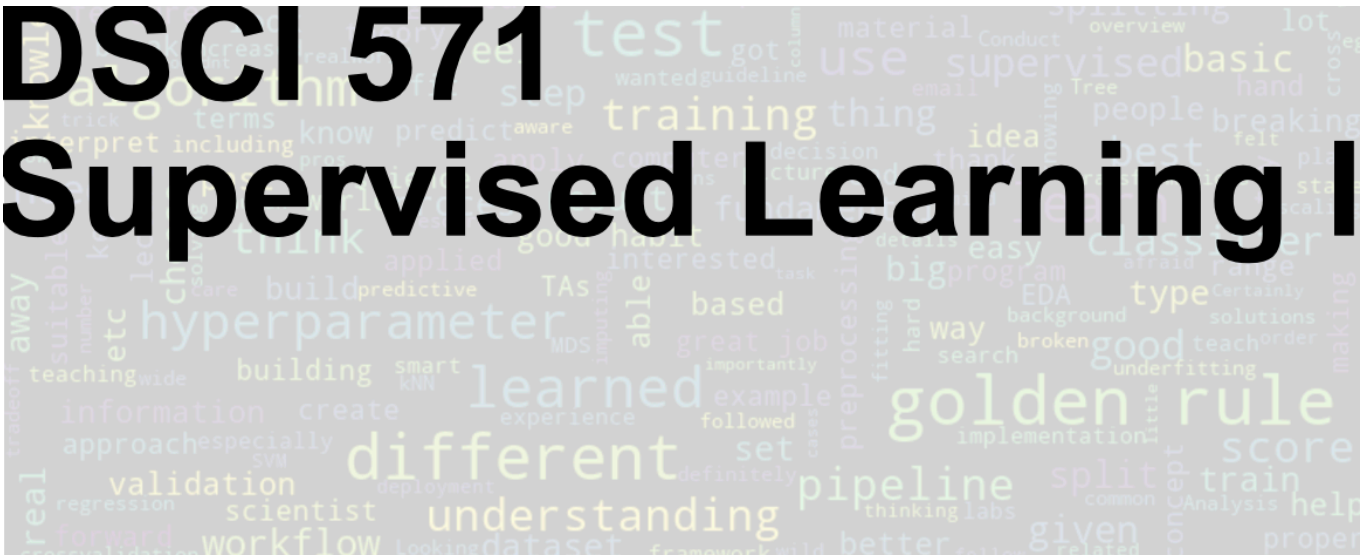
Lecture 5: More preprocessing, text features

Print to PDF

Contents



UBC Master of Data Science program, 2024-25

Imports, Announcements, and LO

Imports

```
import os
import sys

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from IPython.display import HTML

sys.path.append(os.path.join(os.path.abspath(".."), "code"))
from plotting_functions import *
from utils import *

pd.set_option("display.max_colwidth", 200)

from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier, DummyRegressor
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score, cross_validate, train_test
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
DATA_DIR = os.path.join(os.path.abspath(".."), "data/")
```

Learning outcomes

From this lecture, you will be able to

- spot problems with commonly used improper methodologies used in preprocessing;
- explain the difference between ordinal encoding vs one-hot encoding;
- explain possible strategies to deal with categorical variables with too many categories;
- explain why text data needs a different treatment than categorical variables;
- use `scikit-learn`'s `CountVectorizer` to encode text data;
- explain and use different hyperparameters of `CountVectorizer`.
- incorporate text features in a machine learning pipeline

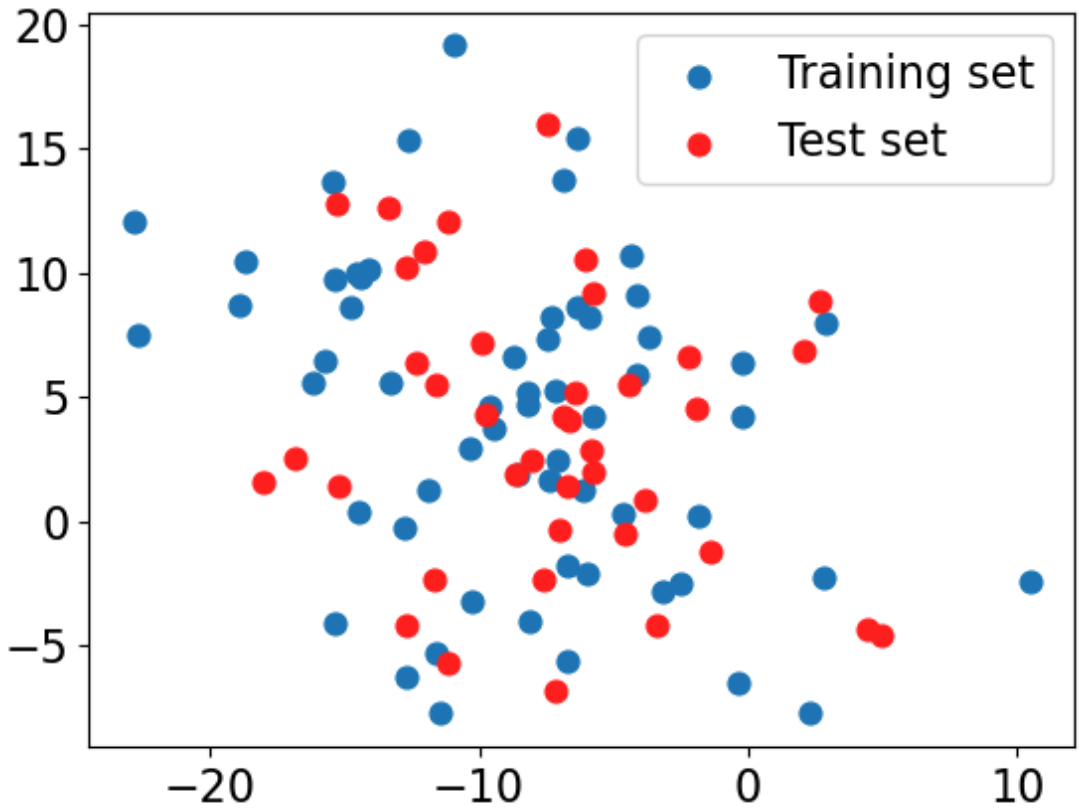
❓ ❓ Questions for you

Let's create some synthetic data.

```
from sklearn.datasets import make_blobs, make_classification

# make synthetic data
X, y = make_blobs(n_samples=100, centers=3, random_state=12, cluster_std=5)
# split it into training and test sets
X_train_toy, X_test_toy, y_train_toy, y_test_toy = train_test_split(
    X, y, random_state=5, test_size=0.4)

plt.scatter(X_train_toy[:, 0], X_train_toy[:, 1], label="Training set", s=60)
plt.scatter(
    X_test_toy[:, 0], X_test_toy[:, 1], color=mglearn.cm2(1), label="Test set",
)
plt.legend(loc="upper right");
```



Let's transform the data using `StandardScaler` and examine how the data looks like.

```
scaler = StandardScaler()
train_transformed = scaler.fit_transform(X_train_toy)
test_transformed = scaler.transform(X_test_toy)
```

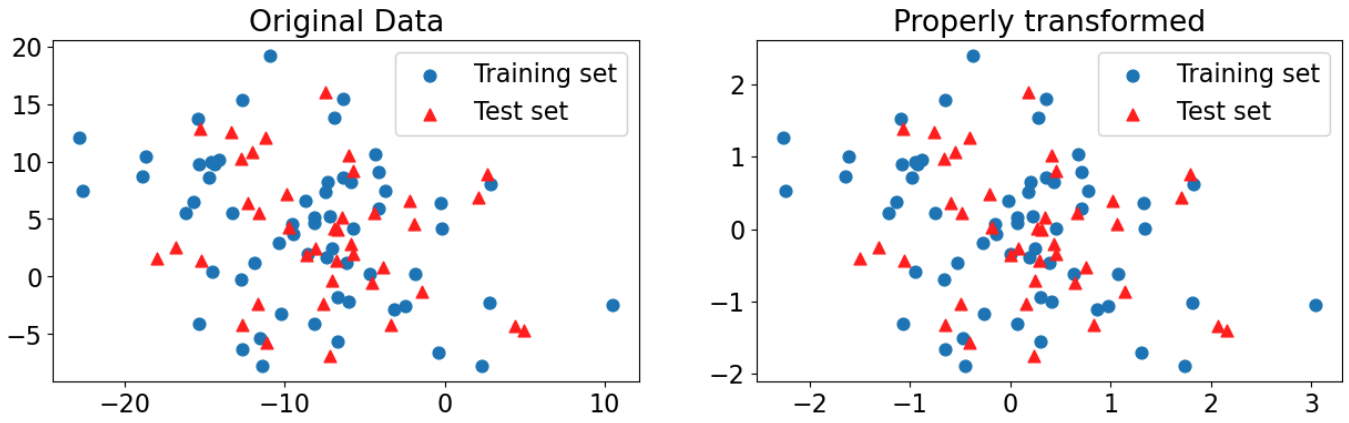
```
X_train_toy[:, 0].mean(), X_train_toy[:, 0].std()
```

```
(np.float64(-8.600287211119548), np.float64(6.270246946519936))
```

```
X_train_toy[:, 1].mean(), X_train_toy[:, 1].std()
```

```
(np.float64(4.124686209266641), np.float64(6.2749592279985595))
```

```
plot_original_scaled(X_train_toy, X_test_toy, train_transformed, test_transform
```




```
knn = KNeighborsClassifier()
knn.fit(train_transformed, y_train_toy)
print(f"Training score: {knn.score(train_transformed, y_train_toy):.2f}")
print(f"Test score: {knn.score(test_transformed, y_test_toy):.2f}")
```

```
Training score: 0.63
Test score: 0.55
```

Bad methodology 1: Scaling the data separately (for class discussion)

```
# DO NOT DO THIS! For illustration purposes only.
scaler = StandardScaler()
scaler.fit(X_train_toy)
train_scaled = scaler.transform(X_train_toy)

scaler = StandardScaler()  # Creating a separate object for scaling test data
scaler.fit(X_test_toy)  # Calling fit on the test data
test_scaled = scaler.transform(
    X_test_toy
)  # Transforming the test data using the scaler fit on test data

knn = KNeighborsClassifier()
knn.fit(train_scaled, y_train_toy)
print(f"Training score: {knn.score(train_scaled, y_train_toy):.2f}")
print(f"Test score: {knn.score(test_scaled, y_test_toy):.2f}")
```

```
Training score: 0.63
Test score: 0.60
```

- Is anything wrong in methodology 1? If yes, what is it?

- What are the mean and standard deviation of columns in `X_train_toy` and `X_test_toy`?

```
X_train_toy[:, 0].mean(), X_train_toy[:, 0].std() # mean and std of column 1 ir
```

```
(np.float64(-8.600287211119548), np.float64(6.270246946519936))
```

```
X_train_toy[:, 1].mean(), X_train_toy[:, 1].std() # mean and std of column 2 ir
```

```
(np.float64(4.124686209266641), np.float64(6.2749592279985595))
```

What are the mean and standard deviation of columns in `X_test_toy`?

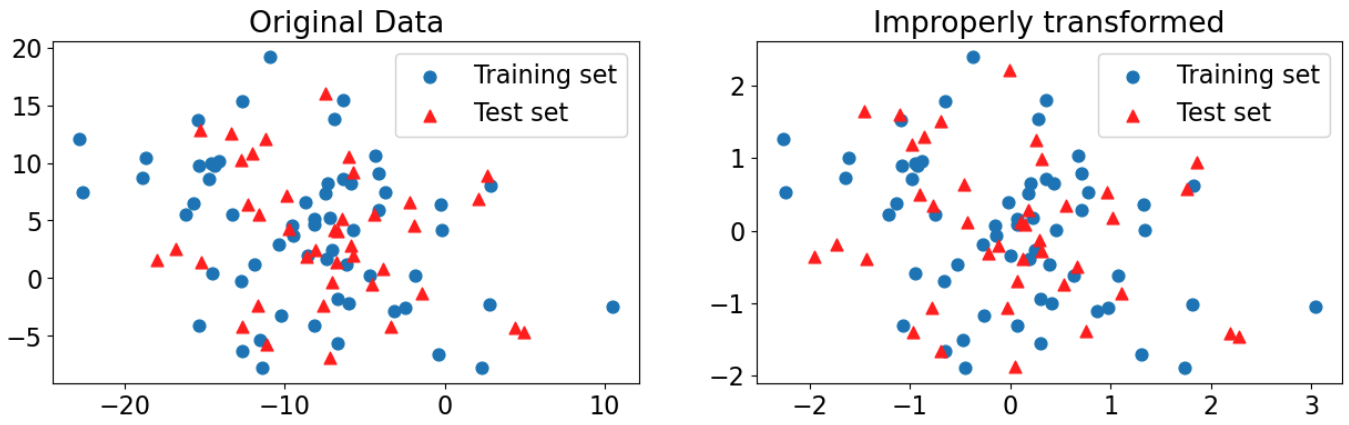
```
X_test_toy[:, 0].mean(), X_test_toy[:, 0].std() # mean and std of column 1 in X
```

```
(np.float64(-7.4360796317881865), np.float64(5.415791218175097))
```

```
X_test_toy[:, 1].mean(), X_test_toy[:, 1].std() # mean and std of column 2 in X
```

```
(np.float64(3.584450392629585), np.float64(5.5947435906433025))
```

```
plot_original_scaled(
    X_train_toy,
    X_test_toy,
    train_scaled,
    test_scaled,
    title_transformed="Improperly transformed",
)
```



Bad methodology 2: Scaling the data together (for class discussion)

```
X_train_toy.shape, X_test_toy.shape
```

```
((60, 2), (40, 2))
```

```
# join the train and test sets back together
XX = np.vstack((X_train_toy, X_test_toy))
XX.shape
```

```
(100, 2)
```

```
scaler = StandardScaler()
scaler.fit(XX)
XX_scaled = scaler.transform(XX)
XX_train = XX_scaled[:X_train_toy.shape[0]]
XX_test = XX_scaled[X_train_toy.shape[0]:]
```

```
knn = KNeighborsClassifier()
knn.fit(XX_train, y_train_toy)
print(f"Training score: {knn.score(XX_train, y_train_toy):.2f}")  # Misleading
print(f"Test score: {knn.score(XX_test, y_test_toy):.2f}")  # Misleading score
```

```
Training score: 0.63
Test score: 0.55
```

- Is anything wrong in methodology 2? If yes, what is it?
- What's are the mean and std of `X_train_toy` vs. `XX`?

```
X_train_toy[:, 0].mean(), X_train_toy[:, 0].std() # mean and std of column 1 ir
```

```
(np.float64(-8.600287211119548), np.float64(6.270246946519936))
```

```
X_train_toy[:, 1].mean(), X_train_toy[:, 1].std() # mean and std of column 2 ir
```

```
(np.float64(4.124686209266641), np.float64(6.2749592279985595))
```

What are the mean and standard deviation of columns in `XX`?

```
XX[:, 0].mean(), XX[:, 0].std() # mean and std of column 1 in XX
```

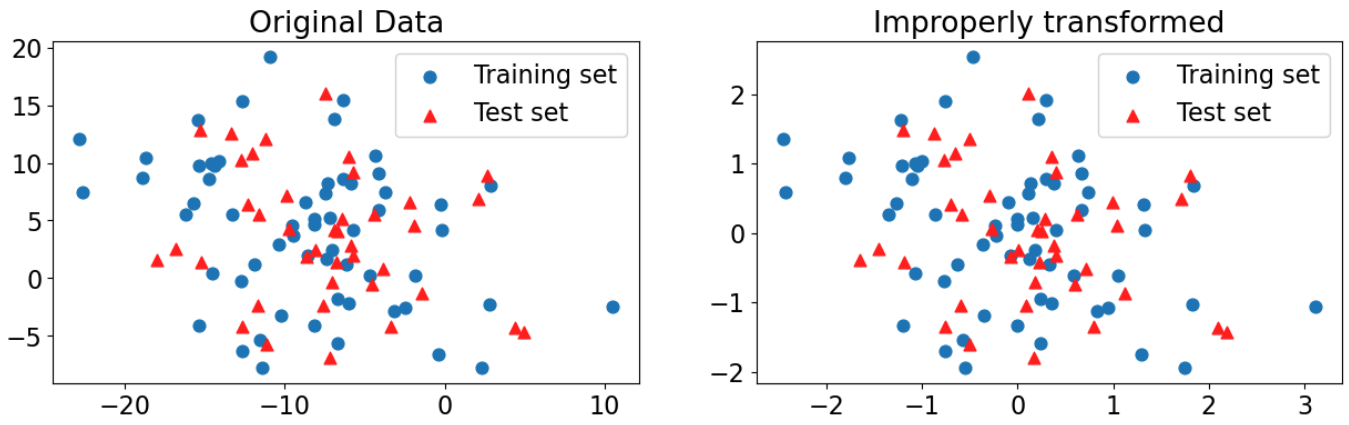
```
(np.float64(-8.134604179387004), np.float64(5.970528192615454))
```

```
XX[:, 1].mean(), XX[:, 1].std() # mean and std of column 2 in XX
```

```
(np.float64(3.9085918826118187), np.float64(6.017937808158447))
```

There is no big difference but they are not the same.

```
plot_original_scaled(
    X_train_toy,
    X_test_toy,
    XX_train,
    XX_test,
    title_transformed="Improperly transformed",
)
```



Not a noticeable difference in the transformed data and in scores in this case. But there is information leakage which is not good.

More discussion on preprocessing

Let's bring back our quiz2 grades toy dataset.

```
grades_df = pd.read_csv(DATA_DIR + 'quiz2-grade-toy-col-transformer.csv')
grades_df.head()
```

	enjoy_course	ml_experience	major	class_attendance	university_years
0	yes	1	Computer Science	Excellent	3
1	yes	1	Mechanical Engineering	Average	2
2	yes	0	Mathematics	Poor	3
3	no	0	Mathematics	Excellent	3
4	yes	0	Psychology	Good	4

```
X, y = grades_df.drop(columns=['quiz2']), grades_df['quiz2']
```

```
numeric_feats = ["university_years", "lab1", "lab3", "lab4", "quiz1"]  # apply
categorical_feats = ["major"]  # apply one-hot encoding
passthrough_feats = ["ml_experience"]  # do not apply any transformation
drop_feats = [
    "lab2",
    "class_attendance",
    "enjoy_course",
]  # do not include these features in modeling
```

One-hot encoding

- What's the purpose of the following arguments of one-hot encoding?
  - handle_unknown="ignore"
  - sparse_output=False

Categorical features with only two possible categories

- Sometimes you have features with only two possible categories.
- If we apply `OheHotEncoder` on such columns, it'll create two columns, which seems wasteful, as we could represent all information in the column in just one column with say 0's and 1's with presence of absence of one of the categories.
- You can pass `drop="if_binary"` argument to `OneHotEncoder` in order to create only one column in such scenario.

```
X["enjoy_course"].head()
```

```
0    yes
1    yes
2    yes
3     no
4    yes
Name: enjoy_course, dtype: object
```

```
ohe_enc = OneHotEncoder(drop="if_binary", dtype=int, sparse_output=False)
ohe_enc.fit(X[["enjoy_course"]])
transformed = ohe_enc.transform(X[["enjoy_course"]])
df = pd.DataFrame(data=transformed, columns=["enjoy_course_enc"], index=X.index
pd.concat([X[["enjoy_course"]], df], axis=1).head(10)
```

	enjoy_course	enjoy_course_enc
0	yes	1
1	yes	1
2	yes	1
3	no	0
4	yes	1
5	no	0
6	yes	1
7	no	0
8	no	0
9	yes	1

Ordinal encoding (class discussion)

- What's the difference between ordinal encoding and one-hot encoding?
- What happens if we do not order the categories when we apply ordinal encoding? Does it matter if we order the categories in ascending or descending order?
- What would happen if an unknown category shows up during validation or test time during ordinal encoding? For example, for `class_attendance` feature what if a category called "super poor" shows up?

More than one ordinal columns?

- We can pass the manually ordered categories when we create an `OrdinalEncoder` object as a list of lists.
- If you have more than one ordinal columns
    - manually create a list of ordered categories for each column

- pass a list of lists to `OrdinalEncoder`, where each inner list corresponds to manually created list of ordered categories for a corresponding ordinal column.

Discussion question

Since `enjoy_course` feature is binary you decide to apply one-hot encoding with `drop="if_binary"`. Your friend decide to apply ordinal encoding on it. Will it make any difference in the transformed data?

```
ohe = OneHotEncoder(drop="if_binary", sparse_output=False)
ohe_encoded = ohe.fit_transform(grades_df[['enjoy_course']]).ravel()
```

```
oe = OrdinalEncoder()
oe_encoded = oe.fit_transform(grades_df[['enjoy_course']]).ravel()
```

```
data = { "oe_encoded": oe_encoded,
         "ohe_encoded": ohe_encoded}
pd.DataFrame(data)
```

	oe_encoded	ohe_encoded
0	1.0	1.0
1	1.0	1.0
2	1.0	1.0
3	0.0	0.0
4	1.0	1.0
5	0.0	0.0
6	1.0	1.0
7	0.0	0.0
8	0.0	0.0
9	1.0	1.0
10	1.0	1.0
11	1.0	1.0
12	1.0	1.0
13	1.0	1.0
14	0.0	0.0
15	0.0	0.0
16	1.0	1.0
17	1.0	1.0
18	0.0	0.0
19	0.0	0.0
20	1.0	1.0

Cases where it's OK to break the golden rule

- If it's some fix number of categories. For example, if it's something like provinces in Canada or majors taught at UBC. We know the categories in advance and this is one of the cases where it might be OK to violate the golden rule and pass the list of known/possible categories. It's OK to incorporate human knowledge in the model.

OHE with many categories

- Do we have enough data for rare categories to learn anything meaningful?
- How about grouping them into bigger categories?
  - Example: country names into continents such as “South America” or “Asia”
- How about considering the most frequent categories and having “other” category for rare cases?

Do we actually want to use certain features for prediction?

- Do you want to use certain features such as **gender** or **race** in prediction?
- Remember that the systems you build are going to be used in some applications.
- It’s extremely important to be mindful of the consequences of including certain features in your predictive model.

Discretizing

- Sometimes you want to transform numeric features into categorical features. This is called discretizing, bucketing, or binning.
- Example: You might want to group ages into categories like *children*, *teenager*, *young adults*, *middle-aged*, and *seniors* for easier interpretation or to maintain privacy, or to capture non-linear relationships in linear models.
- In `sklearn` you can do this using `KBinsDiscretizer` transformer.

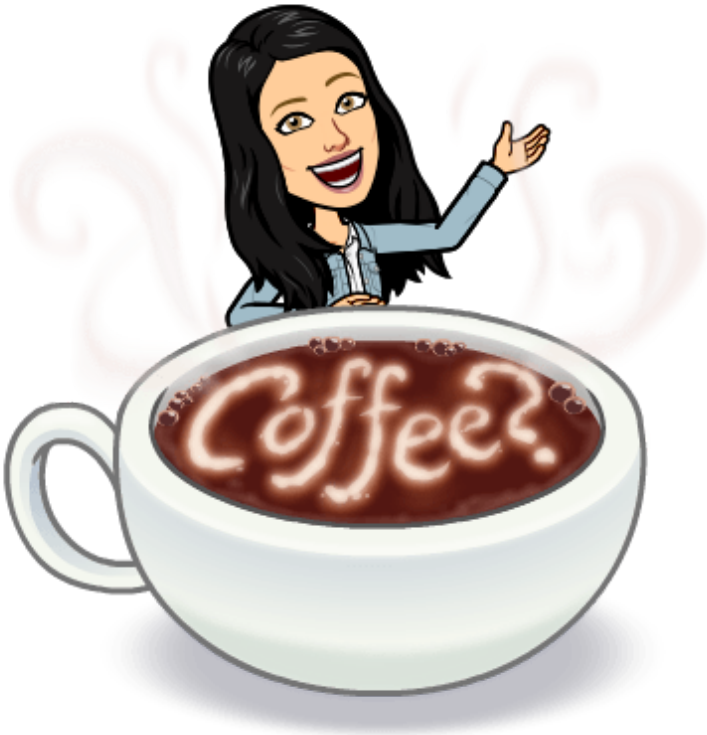
Preprocessing the targets?

- Generally no need for this when doing classification.
- `sklearn` is fine with categorical labels (y-values) for classification problems.
- But if you are using something outside `sklearn`, you might want to apply `LabelEncoder` on the target.
- In regression it makes sense in some cases. More on this in DSCI 573.

Interim summary

- We have only scratched the surface of preprocessing to get us going.
- In general, there is no one perfect solution to preprocessing.
- You as a machine learning practitioner need to carefully look at the data, make appropriate decisions, and justify your decisions in the given situation. – You are likely to follow the steps below in the process.
    - Have a long conversation with the stakeholder(s) who will be using your pipeline.
    - Have a long conversation with the person(s) who collected the data.
    - Think about the ethical implications - are you sure you want to do this project? If so, should ethics guide your approach?

Break (5 min)



So far we talked about data where

- we have a fixed number of features

- features are either continuous or categorical

There are many types of data where there are no predefined features or fixed length descriptions.

- free text data
- image data
- audio data
- video data

In such cases, we need to come up with some representation or fixed length description which is useful for our task.

Let's focus on text data. Let's look at some example reviews from IMDB.

```
imdb_df = pd.read_csv(DATA_DIR + "imdb_master.csv", encoding="ISO-8859-1")
imdb_df['review'].loc[10302]
```

```
'It is a story of Siberian village people from the beginning of 20th century ti
```

```
imdb_df['review'].loc[10]
```

```
'Phil the Alien is one of those quirky films where the humour is based around t
```

- This is how the text looks like in the wild.
- Clearly they do not have the same length.
- There is no obvious way to represent these strings as floating point numbers which is required by standard machine learning approaches.
- There are some weird things going on in these texts
    - HTML markup
    - Capitalization
    - Punctuation marks
    - Misspellings
    - Proper names
- How to tokenize?

- How to normalize words?
- What to include in the vocabulary?
  - exclude stopwords (most common words)
  - exclude rarely occurring words

Encoding text data

```
toy_spam = [
    [
        "URGENT!! As a valued network customer you have been selected to receiv
        "spam",
    ],
    ["Lol you are always so convincing.", "non spam"],
    ["Nah I don't think he goes to usf, he lives around here though", "non spam
    [
        "URGENT! You have won a 1 week FREE membership in our £100000 prize Jac
        "spam",
    ],
    [
        "Had your mobile 11 months or more? U R entitled to Update to the lates
        "spam",
    ],
    ["Congrats! I can't wait to see you!!", "non spam"],
]
toy_df = pd.DataFrame(toy_spam, columns=["sms", "target"])
```

Spam/non spam toy example

- What if the feature is in the form of raw text?
- The feature `sms` below is neither categorical nor ordinal.
- How can we encode it so that we can pass it to the machine learning algorithms we have seen so far?

```
toy_df
```

	sms	target
0	URGENT!! As a valued network customer you have been selected to receive a £900 prize reward!	spam
1	Lol you are always so convincing.	non spam
2	Nah I don't think he goes to usf, he lives around here though	non spam
3	URGENT! You have won a 1 week FREE membership in our £100000 prize Jackpot!	spam
4	Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030	spam
5	Congrats! I can't wait to see you!!	non spam

What if we apply OHE?

```
### DO NOT DO THIS.
enc = OneHotEncoder(sparse_output=False)
transformed = enc.fit_transform(toy_df[["sms"]])
pd.DataFrame(transformed, columns=enc.categories_)
```

	Congrats! I can't wait to see you!!	Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030	Lol you are always so convincing.	Nah I don't think he goes to usf, he lives around here though	URGENT! You have won a 1 week FREE membership in our £100000 prize Jackpot!	URGENT!! As a valued network customer you have been selected to receive a £900 prize reward!
0	0.0	0.0	0.0	0.0	0.0	1.0
1	0.0	0.0	1.0	0.0	0.0	0.0
2	0.0	0.0	0.0	1.0	0.0	0.0
3	0.0	0.0	0.0	0.0	1.0	0.0
4	0.0	1.0	0.0	0.0	0.0	0.0
5	1.0	0.0	0.0	0.0	0.0	0.0

- We do not have a fixed number of categories here.
- Each “category” (feature value) is likely to occur only once in the training data and we won’t learn anything meaningful if we apply one-hot encoding or ordinal encoding on this feature.

- How can we encode or represent raw text data into fixed number of features so that we can learn some useful patterns from it?
- This is a well studied problem in the field of Natural Language Processing (NLP), which is concerned with giving computers the ability to understand written and spoken language.
- Some popular representations of raw text include:
  - **Bag of words**
  - TF-IDF
  - Embedding representations

Bag of words (BOW) representation

- One of the most popular representation of raw text
- Ignores the syntax and word order

- It has two components:
  - The vocabulary (all unique words in all documents)
  - A value indicating either the presence or absence or the count of each word in the document.

Source

Extracting BOW features using `scikit-learn`

- `CountVectorizer`
  - Converts a collection of text documents to a matrix of word counts.
  - Each row represents a "document" (e.g., a text message in our example).
  - Each column represents a word in the vocabulary (the set of unique words) in the training data.
  - Each cell represents how often the word occurs in the document.

> **Note**
>
> In the Natural Language Processing (NLP) community text data is referred to as a **corpus** (plural: corpora). ```

```
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X_counts = vec.fit_transform(toy_df["sms"])
bow_df = pd.DataFrame(
    X_counts.toarray(), columns=vec.get_feature_names_out(), index=toy_df["sms"
)
bow_df
```

	08002986030	100000	11	900	always	are	around	as	bee
sms									
URGENT!! As a valued network customer you have been selected to receive a £900 prize reward!	0	0	0	1	0	0	0	1	
Lol you are always so convincing.	0	0	0	0	1	1	0	0	
Nah I don't think he goes to usf, he lives around here though	0	0	0	0	0	0	1	0	
URGENT! You have won a 1 week FREE membership in our £100000 prize Jackpot!	0	1	0	0	0	0	0	0	
Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030	1	0	1	0	0	0	0	0	
Congrats! I can't wait to see you!!	0	0	0	0	0	0	0	0	

6 rows × 61 columns

```
type(toy_df["sms"])
```

```
pandas.core.series.Series
```

Important

Note that unlike other transformers we are passing a `Series` object to `fit_transform`. For other transformers, you can define one transformer for more than one columns. But with `CountVectorizer` you need to define separate `CountVectorizer` transformers for each text column, if you have more than one text columns.

```
X_counts
```

```
<Compressed Sparse Row sparse matrix of dtype 'int64'
        with 71 stored elements and shape (6, 61)>
```

Why sparse matrices?

- Most words do not appear in a given document.
- We get massive computational savings if we only store the nonzero elements.
- There is a bit of overhead, because we also need to store the locations:
  - e.g. “location (3,27): 1”.
- However, if the fraction of nonzero is small, this is a huge win.

```
print("The total number of elements: ", np.prod(X_counts.shape))
print("The number of non-zero elements: ", X_counts.nnz)
print(
    "Proportion of non-zero elements: %0.4f" % (X_counts.nnz / np.prod(X_counts
)
print(
    "The value at cell 3,%d is: %d"
    % (vec.vocabulary_["jackpot"], X_counts[3, vec.vocabulary_["jackpot"]])
)
```

```
The total number of elements:  366
The number of non-zero elements:  71
Proportion of non-zero elements: 0.1940
The value at cell 3,27 is: 1
```

Question for you

- What would happen if you apply `StandardScaler` on sparse data?

`OneHotEncoder` and sparse features

- By default, `OneHotEncoder` also creates sparse features.
- You could set `sparse=False` to get a regular `numpy` array.
- If there are a huge number of categories, it may be beneficial to keep them sparse.
- For smaller number of categories, it doesn't matter much.

Important hyperparameters of `CountVectorizer`

- `binary`
  - whether to use absence/presence feature values or counts
- `max_features`
  - only consider top `max_features` ordered by frequency in the corpus
- `max_df`
  - ignore features which occur in more than `max_df` documents
- `min_df`
  - ignore features which occur in less than `min_df` documents
- `ngram_range`
  - consider word sequences in the given range

Let's look at all features, i.e., words (along with their frequencies).

```
vec = CountVectorizer()
X_counts = vec.fit_transform(toy_df["sms"])
bow_df = pd.DataFrame(
    X_counts.toarray(), columns=vec.get_feature_names_out(), index=toy_df["sms"
)
bow_df
```

	08002986030	100000	11	900	always	are	around	as	bee
sms									
URGENT!! As a valued network customer you have been selected to receive a £900 prize reward!	0	0	0	1	0	0	0	1	
Lol you are always so convincing.	0	0	0	0	1	1	0	0	
Nah I don't think he goes to usf, he lives around here though	0	0	0	0	0	0	1	0	
URGENT! You have won a 1 week FREE membership in our £100000 prize Jackpot!	0	1	0	0	0	0	0	0	
Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030	1	0	1	0	0	0	0	0	
Congrats! I can't wait to see you!!	0	0	0	0	0	0	0	0	

6 rows × 61 columns

When we use `binary=True`, the representation uses presence/absence of words instead of word counts.

```
vec_binary = CountVectorizer(binary=True)
X_counts = vec_binary.fit_transform(toy_df["sms"])
bow_df = pd.DataFrame(
    X_counts.toarray(), columns=vec_binary.get_feature_names_out(), index=toy_d
)
bow_df
```

	08002986030	100000	11	900	always	are	around	as	bee
sms									
URGENT!! As a valued network customer you have been selected to receive a £900 prize reward!	0	0	0	1	0	0	0	1	
Lol you are always so convincing.	0	0	0	0	1	1	0	0	
Nah I don't think he goes to usf, he lives around here though	0	0	0	0	0	0	1	0	
URGENT! You have won a 1 week FREE membership in our £100000 prize Jackpot!	0	1	0	0	0	0	0	0	
Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030	1	0	1	0	0	0	0	0	
Congrats! I can't wait to see you!!	0	0	0	0	0	0	0	0	

6 rows × 61 columns

We can control the size of X (the number of features) using `max_features`.

```
vec8 = CountVectorizer(max_features=8)
X_counts = vec8.fit_transform(toy_df["sms"])
bow_df = pd.DataFrame(
    X_counts.toarray(), columns=vec8.get_feature_names_out(), index=toy_df["sms
)
bow_df
```

	free	have	mobile	the	to	update	urgent	you
sms								
URGENT!! As a valued network customer you have been selected to receive a £900 prize reward!	0	1	0	0	1	0	1	1
Lol you are always so convincing.	0	0	0	0	0	0	0	1
Nah I don't think he goes to usf, he lives around here though	0	0	0	0	1	0	0	0
URGENT! You have won a 1 week FREE membership in our £100000 prize Jackpot!	1	1	0	0	0	0	1	1
Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030	2	0	2	2	2	2	0	0
Congrats! I can't wait to see you!!	0	0	0	0	1	0	0	1

Note

Notice that `vec8` and `vec8_binary` have different vocabularies, which is kind of unexpected behaviour and doesn't match the documentation of `scikit-learn`.

Here is the code for `binary=True` condition in `scikit-learn`. As we can see, the binarization is done before limiting the features to `max_features`, and so now we are actually looking at the document counts (in how many documents it occurs) rather than term count. This is not explained anywhere in the documentation.

The ties in counts between different words makes it even more confusing. I don't think it'll have a big impact on the results but this is good to know! Remember that `scikit-learn` developers are also humans who are prone to make mistakes. So it's always a good habit to question whatever tools we use every now and then.

```
vec8 = CountVectorizer(max_features=8)
X_counts = vec8.fit_transform(toy_df["sms"])
pd.DataFrame(
    data=X_counts.sum(axis=0).tolist()[0],
    index=vec8.get_feature_names_out(),
    columns=["counts"],
).sort_values("counts", ascending=False)
```

	counts
to	5
you	4
free	3
have	2
mobile	2
the	2
update	2
urgent	2

```
vec8_binary = CountVectorizer(binary=True, max_features=8)
X_counts = vec8_binary.fit_transform(toy_df["sms"])
pd.DataFrame(
    data=X_counts.sum(axis=0).tolist()[0],
    index=vec8_binary.get_feature_names_out(),
    columns=["counts"],
).sort_values("counts", ascending=False)
```

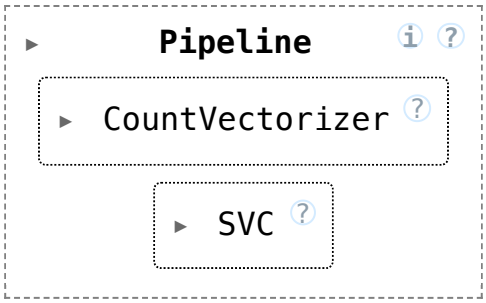
	counts
to	4
you	4
free	2
have	2
prize	2
urgent	2
mobiles	1
months	1

Preprocessing

- Note that `CountVectorizer` is carrying out some preprocessing such as because of the default argument values
  - Converting words to lowercase (`lowercase=True`)
  - getting rid of punctuation and special characters (`token_pattern='(?u)\\b\\w\\w+\\b'`)

```
pipe = make_pipeline(CountVectorizer(), SVC())
```

```
pipe.fit(toy_df["sms"], toy_df["target"])
```




```
pipe.predict(toy_df["sms"])
```

```
array(['spam', 'non spam', 'non spam', 'spam', 'spam', 'non spam'],
      dtype=object)
```

Is this a realistic representation of text data?

- Of course this is not a great representation of language
  - We are throwing out everything we know about language and losing a lot of information.
  - It assumes that there is no syntax and compositional meaning in language.
- But it works surprisingly well for many tasks.
- We will learn more expressive representations in the coming weeks.

(Optional) tf-idf representation

- Another common way to represent text is using tf-idf representation, which stands for term frequency inverse document frequency.
- This is an alternative to using using stop words or putting more emphasis on rare or more specific words.
- If something appears in most of the documents then the inverse document frequency gives it a lower weight.
- This is commonly used in information retrieval, which is the field of finding relevant documents. But people also find it useful in machine learning.

$$\text{tf-idf}(t,d) = tf(t,d).\text{idf}(t,d)$$

$$\text{idf}(t,d) = log\frac{1+n_d}{1+df(d,t)} + 1$$

where

- $n_d \rightarrow$ total number of documents
- $df(d,t) \rightarrow$ number of documents containing the term t
- How to do this in `sklearn`? There are two ways:
  - Using TfidfVectorizer directly
  - Creating a pipeline with `CountVectorizer` and `TfidfTransformer`

```
from sklearn.feature_extraction.text import TfidfVectorizer, TfidfTransformer
tfidf1 = TfidfVectorizer()
X_tfidf1 = tfidf1.fit_transform(toy_df["sms"])
pd.DataFrame(X_tfidf1.toarray(), columns = tfidf1.get_feature_names_out().tolis
```

	08002986030	100000	11	900	always	are	around
0	0.000000	0.000000	0.000000	0.292088	0.000000	0.000000	0.00000
1	0.000000	0.000000	0.000000	0.000000	0.432259	0.432259	0.00000
2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.27367
3	0.000000	0.315571	0.000000	0.000000	0.000000	0.000000	0.00000
4	0.173821	0.000000	0.173821	0.000000	0.000000	0.000000	0.00000
5	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000

6 rows × 61 columns

```
tfidf2 = make_pipeline(CountVectorizer(), TfidfTransformer())
X_tfidf2 = tfidf2.fit_transform(toy_df["sms"])
pd.DataFrame(X_tfidf2.toarray(), columns = tfidf2.get_feature_names_out().tolis
```

	08002986030	100000	11	900	always	are	around
0	0.000000	0.000000	0.000000	0.292088	0.000000	0.000000	0.00000
1	0.000000	0.000000	0.000000	0.000000	0.432259	0.432259	0.00000
2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.27367
3	0.000000	0.315571	0.000000	0.000000	0.000000	0.000000	0.00000
4	0.173821	0.000000	0.173821	0.000000	0.000000	0.000000	0.00000
5	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000

6 rows × 61 columns

By default, `sklearn` carries out L2 normalization. In other words, it divides each value by the Euclidean norm of that row.

❓❓ Questions for you

(iClicker) Exercise 5.1

Select all of the following statements which are TRUE.

- (A) `handle_unknown="ignore"` would treat all unknown categories equally.
- (B) As you increase the value for `max_features` hyperparameter of `CountVectorizer` the training score is likely to go up.
- (C) Suppose you are encoding text data using `CountVectorizer`. If you encounter a word in the validation or the test split that's not available in the training data, we'll get an error.
- (D) In the code below, inside `cross_validate`, each fold might have slightly different number of features (columns) in the fold.

```
pipe = (CountVectorizer(), SVC())
cross_validate(pipe, X_train, y_train)
```

V's Solutions!

Discussion questions

- What's the problem with calling `fit_transform` on the test data in the context of `CountVectorizer`?
- Do we need to scale after applying bag-of-words representation?

What did we learn today?

- More on ordinal features
- Different arguments `OneHotEncoder`
  - `handle_unknow="ignore"`
  - `if_binary`
- Dealing with text features
  - Bag of words representation: `CountVectorizer`


TALK
soon