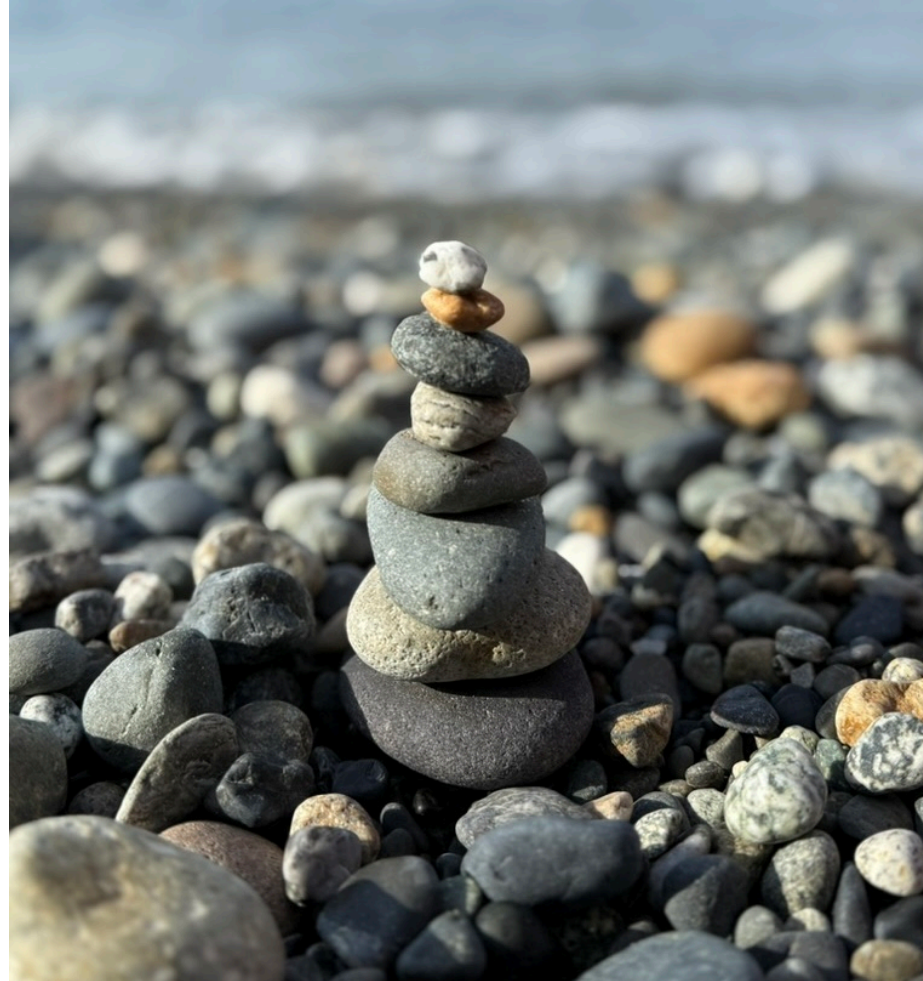


Lecture 6: Column transformer and text features

Varada Kolhatkar

Focus on the breath!



Announcements

- HW3 is due Monday, Oct 5th, 11:59 pm.
 - If you haven't started yet, start now.
 - You can work in pairs for this assignment.
- My OH: Tuesdays and Thursdays right after the class.

Today's big questions

- Which **preprocessing mistakes** make our evaluation unreliable?
- How do we apply **different transformations to different columns**?
- How can we represent **text as numerical features**?

Mistake 1: fitting train and test separately

What's wrong here? Assume we have already split the data.

```
1 train_scaler = StandardScaler()  
2 train_scaled = train_scaler.fit_transform(X_train)  
3  
4 test_scaler = StandardScaler()  
5 test_scaled = test_scaler.fit_transform(X_test)
```

Same age, different coordinates

Suppose **age is our only feature**:

Fit the scaler on...	Ages	Mean	Standard deviation
Training set	20, 40	30	10
Test set	40, 60	50	10

For a **40-year-old**:

- Training scaler: $(40 - 30)/10 = +1$
- Separate test scaler: $(40 - 50)/10 = -1$

Should the same age have different coordinates in training and test?

Should the same age have different coordinates in training and test? 

A test point's location relative to the training points determines its neighbours.

Use the **same fitted transformation** for training and unseen rows.

Mistake 2: fitting on train and test together

What's wrong here?

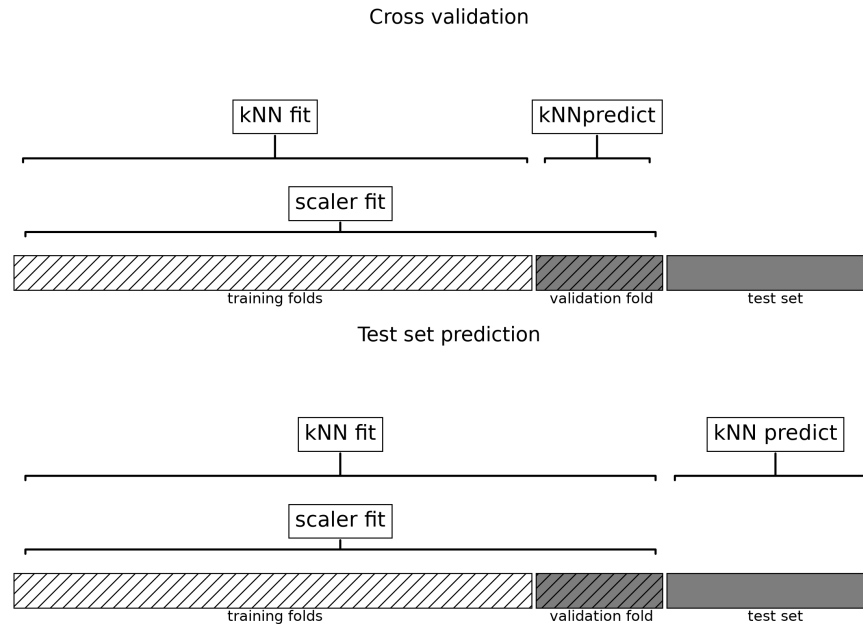
```
1 X_all = np.vstack([X_train, X_test])
2 scaler = StandardScaler().fit(X_all)
3
4 train_scaled = scaler.transform(X_train)
5 test_scaled = scaler.transform(X_test)
```

Mistake 3: fitting before cross-validation

The test set is untouched. Is this evaluation valid?

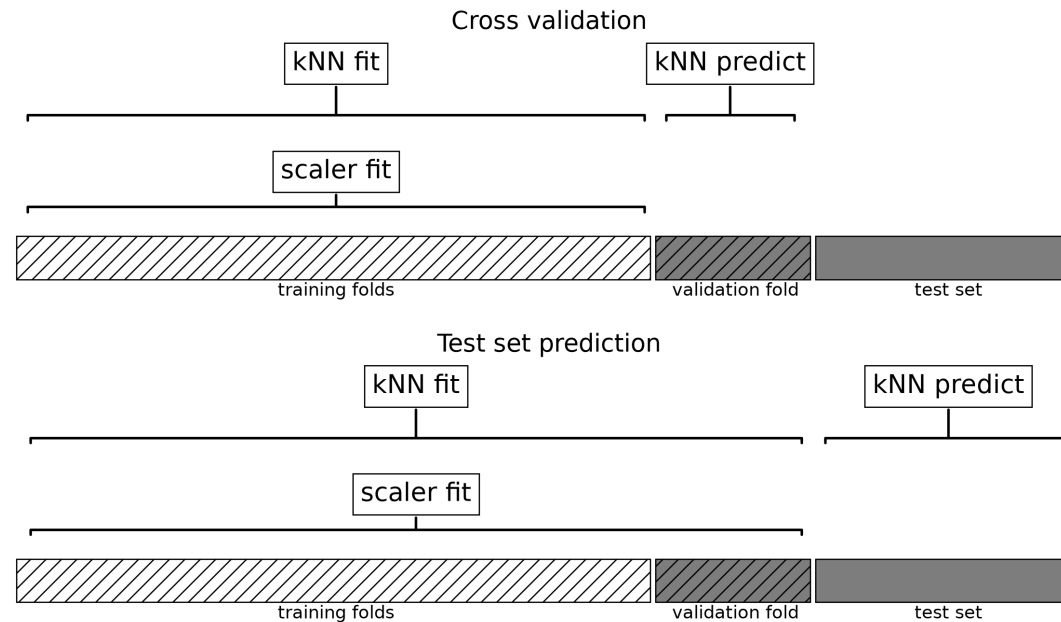
```
1 scaler = StandardScaler()  
2 X_train_scaled = scaler.fit_transform(X_train)  
3  
4 cross_val_score(  
5     KNeighborsClassifier(), X_train_scaled, y_train  
6 )
```

Where the leakage happens



The scaler's fit includes the validation fold. ❌

Which rows may each step learn from?



Within each fold, preprocessing and the model fit on the same training rows. ✓

The repair: cross-validate the whole pipeline

```
1 from sklearn.pipeline import make_pipeline
2 from sklearn.neighbors import KNeighborsClassifier
3 from sklearn.model_selection import cross_val_score
4
5 pipe = make_pipeline(StandardScaler(), KNeighborsClassifier())
6 cross_val_score(pipe, X_train, y_train)
```

- Pass **raw features** and the **whole pipeline** to CV.
- Each fold fits its own preprocessing and classifier.
- Validation rows receive **transform**, then **predict**.

(Optional) Alternative transformations

Method	Transformation
<code>StandardScaler</code>	Centre each feature and scale by its training standard deviation
<code>MinMaxScaler</code>	Map each feature's training range to a chosen interval
<code>Normalizer</code>	Scale each row to unit norm
Log transform	Compress large positive values; can help with skewed counts

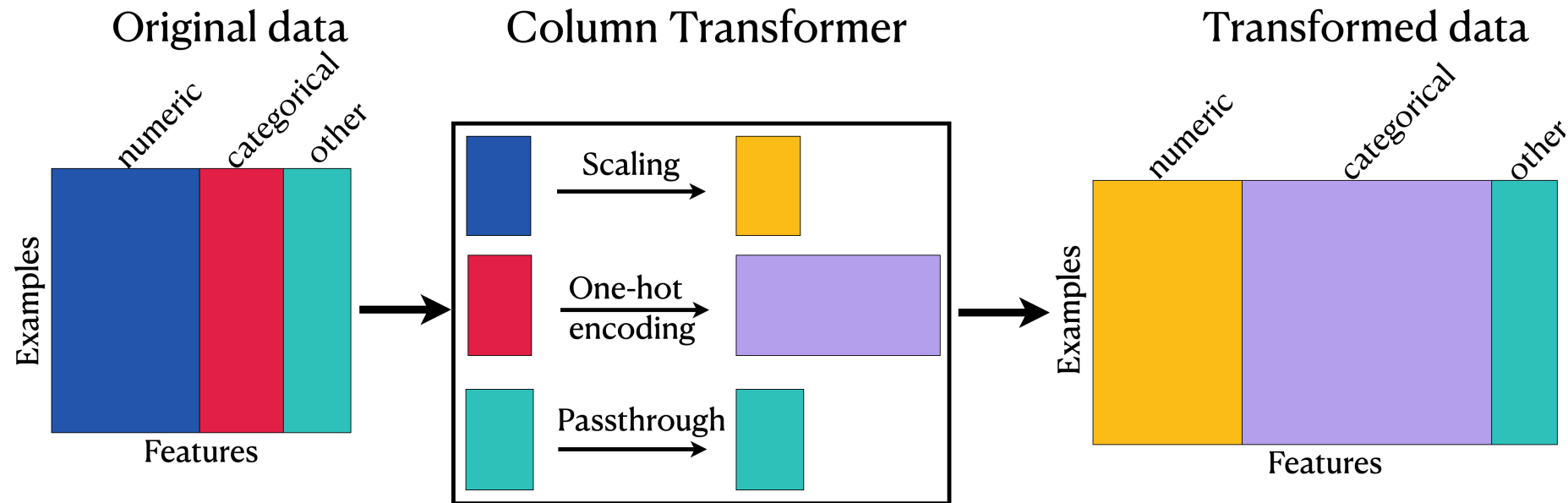
Choose based on the feature and model, then evaluate.

Different columns need different transformations

Feature group	Columns	Preprocessing
Numeric	age, n_people, price	Impute → scale
Unordered categories	food_type, north_america	Impute → one-hot encode
Ordered category	noise_level	Impute → ordinal encode

How do we combine these into one feature matrix?

ColumnTransformer combines feature groups



Different columns → their own transformations → one feature matrix.

Build the numeric and categorical branches

```
1 from sklearn.impute import SimpleImputer
2 from sklearn.preprocessing import StandardScaler, OneHotEncoder
3
4 numeric_pipe = make_pipeline(
5     SimpleImputer(strategy="median"),
6     StandardScaler(),
7 )
8
9 categorical_pipe = make_pipeline(
10     SimpleImputer(strategy="most_frequent"),
11     OneHotEncoder(handle_unknown="ignore"),
12 )
```

Each branch is a **sequence of transformations** for one feature group.

Build the ordinal branch

```
1 from sklearn.preprocessing import OrdinalEncoder
2
3 noise_order = ["no music", "low", "medium", "high", "crazy loud"]
4
5 ordinal_pipe = make_pipeline(
6     SimpleImputer(strategy="most_frequent"),
7     OrdinalEncoder(categories=[noise_order]),
8 )
```

We supply the meaningful order of noise levels.

ColumnTransformer syntax

```
1 from sklearn.compose import ColumnTransformer
2
3 preprocessor = ColumnTransformer([
4     ("num", numeric_pipe, ["age", "n_people", "price"]),
5     ("cat", categorical_pipe, ["food_type", "north_america"]),
6     ("ord", ordinal_pipe, ["noise_level"]),
7 ], remainder="drop")
```

Each tuple contains **(name, transformer, columns)**.

- **Name:** a label for the branch.
- **Transformer:** one transformer or a preprocessing pipeline.
- **Columns:** the input columns for that branch.

The shortcut:

make_column_transformer

```
1 from sklearn.compose import make_column_transformer
2
3 preprocessor = make_column_transformer(
4     (numeric_pipe, ["age", "n_people", "price"]),
5     (categorical_pipe, ["food_type", "north_america"]),
6     (ordinal_pipe, ["noise_level"]),
7     remainder="drop",
8 )
```

Each tuple contains **(transformer, columns)**; names are generated for you.

We'll primarily use this constructor.

What comes out of a column transformer?

- **Same rows:** each output row corresponds to the same input respondent.
- **Output order:** numeric block, categorical block, then ordinal block in our example.
- **Feature count:** may change; one-hot encoding can add columns.
- **Unlisted columns:** dropped by default (`remainder="drop"`).
- **Output format:** may be dense or sparse, depending on the transformations and sparsity.

After fitting, inspect the output names with:

```
1 preprocessor.fit(X_train, y_train)
2 preprocessor.get_feature_names_out()
```

Put preprocessing inside the prediction pipeline

```
1 pipe = make_pipeline(  
2     preprocessor, # column transformer object  
3     KNeighborsClassifier(),  
4 )  
5  
6 scores = cross_val_score(pipe, X_train, y_train, cv=5)
```

Raw survey columns → column transformer → KNN → predictions

Cross-validation fits **every branch and the classifier** within each fold.

Class demo: combine the survey features

[Open the preprocessing demo](#)

Task: infer a respondent's `like` / `dislike` label from their completed survey.

As we build the column transformer:

1. Trace which columns enter each branch.
2. Inspect the transformed feature names and shape.
3. Cross-validate the complete prediction pipeline.

Ordinal or one-hot?

Weather categories: **sunny, cloudy, rainy, snowy**.

Would you impose an order when predicting:

- Traffic volume?
- Severity of weather-related road incidents?

Use an ordinal encoding only if you can justify the order for the feature.

A meaningful order does not automatically make ordinal encoding the right choice. Choose the encoding in the context of the prediction task.

Unseen categories

Suppose `food_type` contains only **Chinese** and **Italian** during training.

With `OneHotEncoder(handle_unknown="ignore")`:

New respondent's <code>food_type</code>	<code>food_type_Chinese</code>	<code>food_type_Italian</code>
Chinese	1	0
Italian	0	1
Ethiopian (unseen)	0	0
Peruvian (unseen)	0	0

Prediction can proceed, but this feature cannot distinguish Ethiopian from Peruvian.

Is this the behaviour you want for your prediction task?

Binary and high-cardinality features

Binary features: `drop="if_binary"` keeps one of the two one-hot columns.

Features with many categories: possible choices include:

- Group rare categories into an “other” category.
- Combine categories using domain knowledge.
- Retain frequent categories and group the rest.

Learn frequency-based grouping rules **inside each CV fold**.

iClicker: Exercise 6.1

Select all statements that are TRUE.

- a. A column transformer alone provides class predictions.
- b. Output feature blocks follow the order of the transformer list.
- c. The number of output features must differ from the number of input features.
- d. A column transformer can be a step in a prediction pipeline.

Text is another feature type

Suppose our restaurant survey includes these comments:

- “good food”
- “good good service”
- “slow service”

How could we turn each review into a row of numbers?

Bag of words: count the vocabulary

Vocabulary: food, good, service, slow

Review	food	good	service	slow
good food	1	1	0	0
good good service	0	2	1	0
slow service	0	0	1	1

- One row per document; one column per vocabulary term.
- Each entry counts occurrences of that term.

CountVectorizer: learn and apply the vocabulary

```
1 from sklearn.feature_extraction.text import CountVectorizer
2
3 reviews = ["good food", "good good service", "slow service"]
4 vectorizer = CountVectorizer()
5 X_counts = vectorizer.fit_transform(reviews)
6
7 vectorizer.get_feature_names_out()
8 # array(['food', 'good', 'service', 'slow'], ...)
```

- **fit**: learn the vocabulary from training documents.
- **transform**: count those terms in each document.
- The count matrix is **sparse**: store nonzero entries efficiently.

Your turn: an unseen review

Training vocabulary: food, good, service, slow

New review: “good excellent food”

What vector do we get? Does `excellent` create a new column?

`[1, 1, 0, 0]`

`excellent` is outside the fitted vocabulary, so it is ignored.

Transform only: keep the learned columns fixed for unseen reviews.

What does bag of words lose?

“the food was good, not bad”

“the food was bad, not good”

These have **the same unigram counts**, but different meanings.

- Word order is lost.
- Counts alone do not capture context or meaning.
- **Bigrams** can retain adjacent pairs such as **not bad** and **not good**.

Choosing the text representation

Parameter	What it controls
<code>max_features</code>	Keep at most this many terms, ranked by corpus frequency
<code>min_df, max_df</code>	Exclude terms based on how many documents contain them
<code>ngram_range=(1, 2)</code>	Include single words and adjacent word pairs
<code>stop_words</code>	Remove specified common words

These are modelling choices to compare using cross-validation.

Removing words can remove meaning: consider “**not good**”.

A pipeline for text classification

```
1 from sklearn.svm import SVC
2
3 text_pipe = make_pipeline(CountVectorizer(), SVC())
4
5 # Training review strings and their corresponding labels
6 cross_val_score(text_pipe, reviews_train, y_train, cv=5)
```

Each fold learns its vocabulary from its training reviews only.

Validation reviews are counted using that vocabulary, then classified.

Combine text with the other columns

```
1 # First replace missing comments with empty strings.
2 X_train = X_train.assign(comments=X_train["comments"].fillna("")
3
4 preprocessor_with_text = make_column_transformer(
5     (numeric_pipe, ["age", "n_people", "price"]),
6     (categorical_pipe, ["food_type", "north_america"]),
7     (ordinal_pipe, ["noise_level"]),
8     (CountVectorizer(), "comments"),
9 )
10 pipe = make_pipeline(preprocessor_with_text, SVC())
```

Use **"comments"**, not **["comments"]**: the vectorizer expects a 1D sequence of strings.

iClicker: Exercise 6.2

Select all statements that are TRUE.

- a. An unseen word causes `CountVectorizer.transform` to fail.
- b. Repeated words can produce counts greater than one.
- c. When cross-validating a pipeline containing `CountVectorizer`, as shown below, each fold may learn a different vocabulary size.

```
1 pipe = make_pipeline(CountVectorizer(), SVC())  
2 cross_val_score(pipe, reviews_train, y_train, cv=5)
```

- d. Increasing `max_features` guarantees a higher validation score.

More preprocessing choices

There is no universal preprocessing recipe.

- Use exploratory analysis and domain knowledge to choose representations.
- Compare choices using cross-validation.
- Keep the test set reserved until those choices are settled.

Summary

- **Preprocessing mistakes:** keep validation and test rows out of every `fit`, including vocabulary learning.
- **Different columns:** use `ColumnTransformer` to combine feature groups inside a prediction pipeline.
- **Text features:** bag of words represents documents using vocabulary counts; it loses word order.

Cross-validate the complete procedure on raw training features.

Exit question

```
1 train_vec = CountVectorizer(max_features=1000)
2 valid_vec = CountVectorizer(max_features=1000)
3
4 X_train = train_vec.fit_transform(reviews_train)
5 X_valid = valid_vec.fit_transform(reviews_valid)
6
7 model = SVC()
8 model.fit(X_train, y_train)
9 model.score(X_valid, y_valid)
```

Both matrices have **1,000 columns**, and the code runs without errors.

Can we trust the score? What would you check or change?

If we remove `max_features=1000`, could `model.score` raise an error? Why?