Project T Final: Data Cleaning

Team .N.A.N.T.S.

Visualizing Data with Matplotlib/Seaborn

Motivation

Data visualization is a task closely tied with data cleaning, since plots and graphs condense large datasets into images that are easily comprehensible by humans.

These visualizations allow us to sanity check our data and model, identify outliers that should be excluded from our training data, and explain our findings to others that might want to learn about our work.

This section discusses good data visualization practices, as well as several basic visualizations.

Best Practices when Creating Visualizations

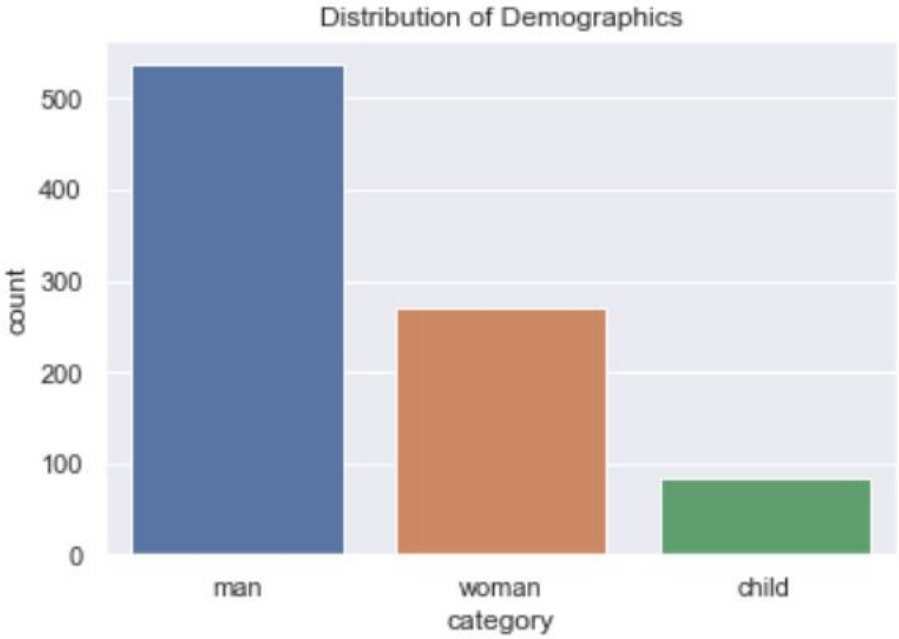
The goal of a visualization is to be easy to read. Graphics should be able to offer clear and accurate information, so that someone viewing it should be able to glean meaningful insights even from a cursory glance. Below is a list of a few things to always do when creating graphics:

- Add descriptive titles and axis labels.
- Use appropriate scaling so that your data is readable.
- Avoid plotting too much data and cluttering your graph (overplotting).
- Consider if you’re plotting categorical or numerical data.

Single Categorical Variable: Bar Chart

Bar charts visualize the distribution of a single categorical variable. The y-axis can be either counts or percentages.

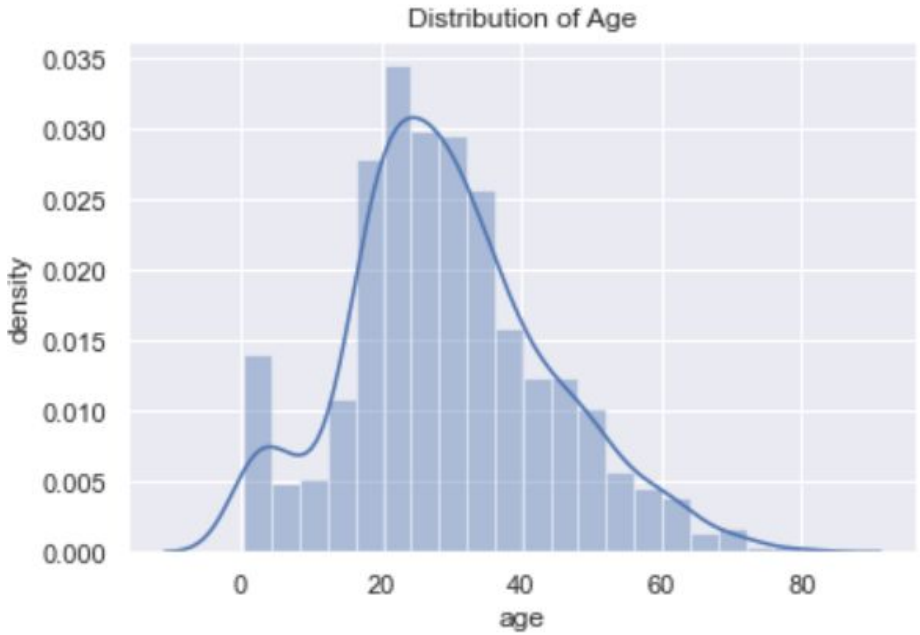
The bar chart on the right shows the distribution of passengers of the Titanic by gender (except child).



Single Numerical Variable: Histogram

Histograms show the distribution of a single numerical variable. Histograms can use either counts or densities for the y-axis units.

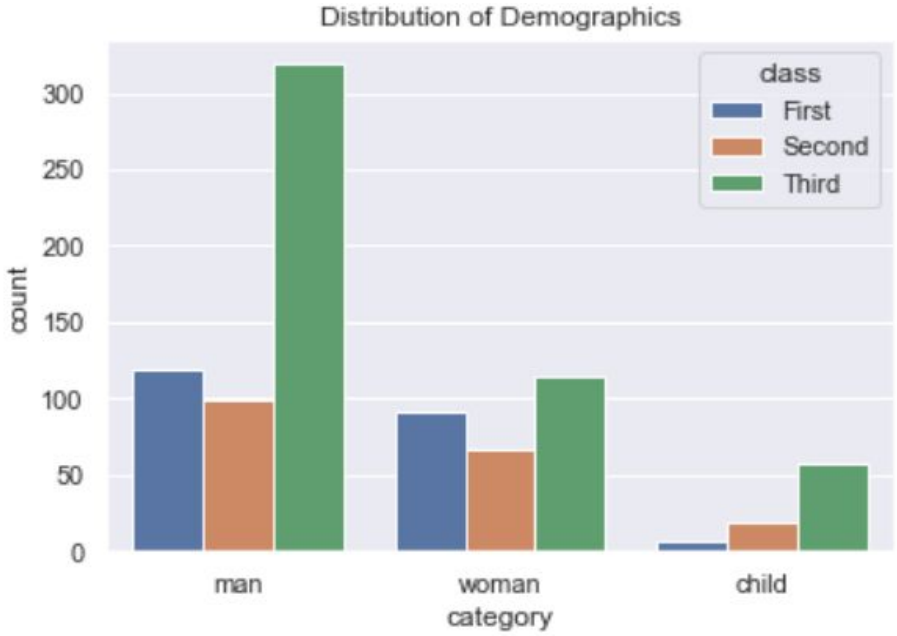
The histogram on the right shows the distribution of the ages of Titanic passengers.



2+ Categorical Variables: Multi-Level Bar Chart

Multi-level bar charts can help visualize multiple categorical variables. Be wary of using this visualization on variables with many labels, as that may compromise readability.

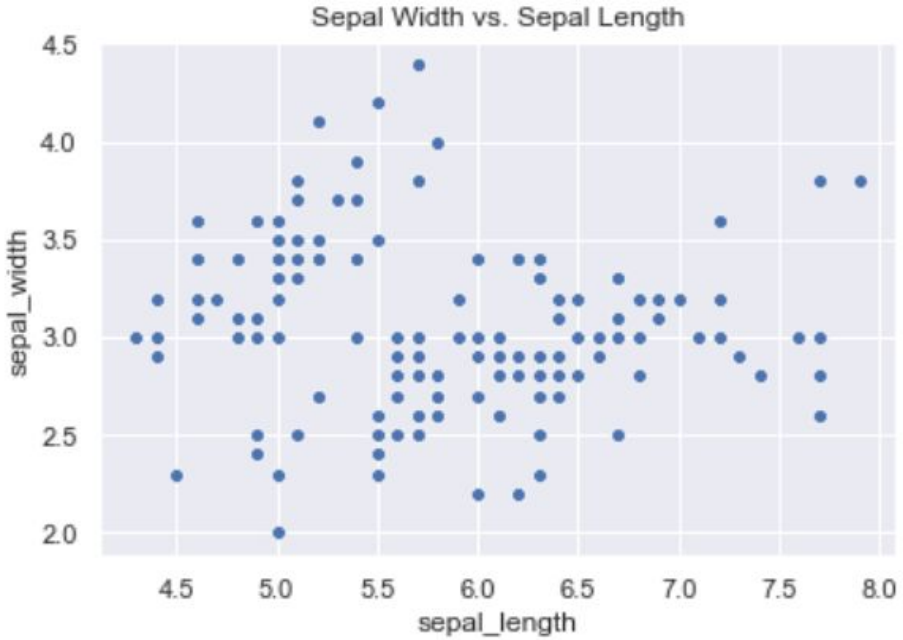
The two-level bar chart on the right shows the distribution of passengers of the Titanic by gender and class.



2 Numerical Variables: Scatter Plot

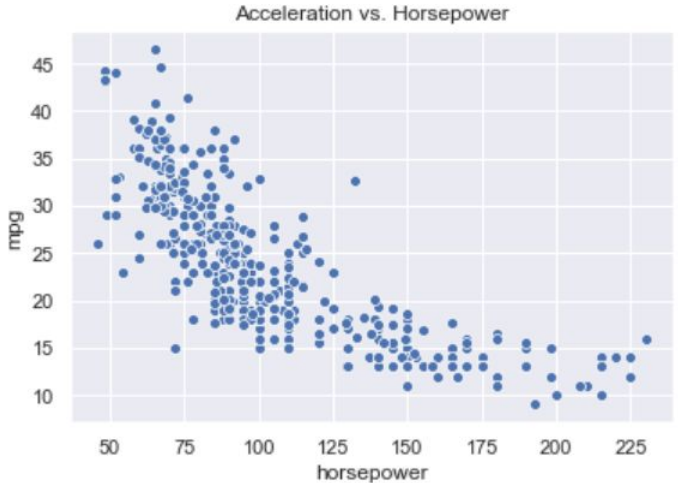
Scatter plots show how two variables are related. While they are powerful visualizations, scatter plots have a few limitations.

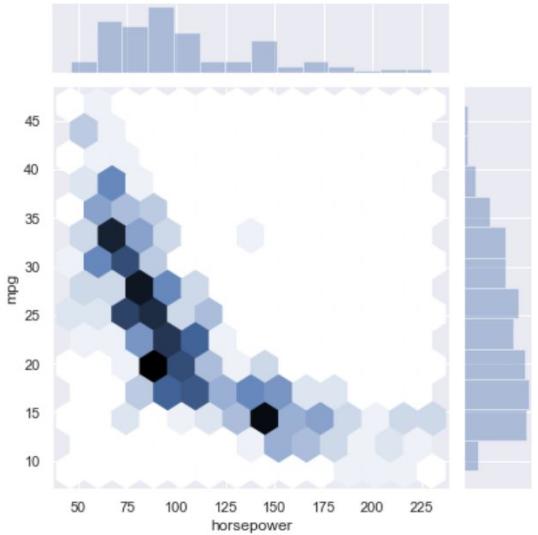
The scatter plot on the right shows how sepal width and sepal length are related in the iris dataset.



2 Numerical Variables: Hex Plot

Hex plots alleviate the problem of overcrowded scatter plots by smoothing out regions. The plots below are a bad scatter plot and its corresponding hex plot:

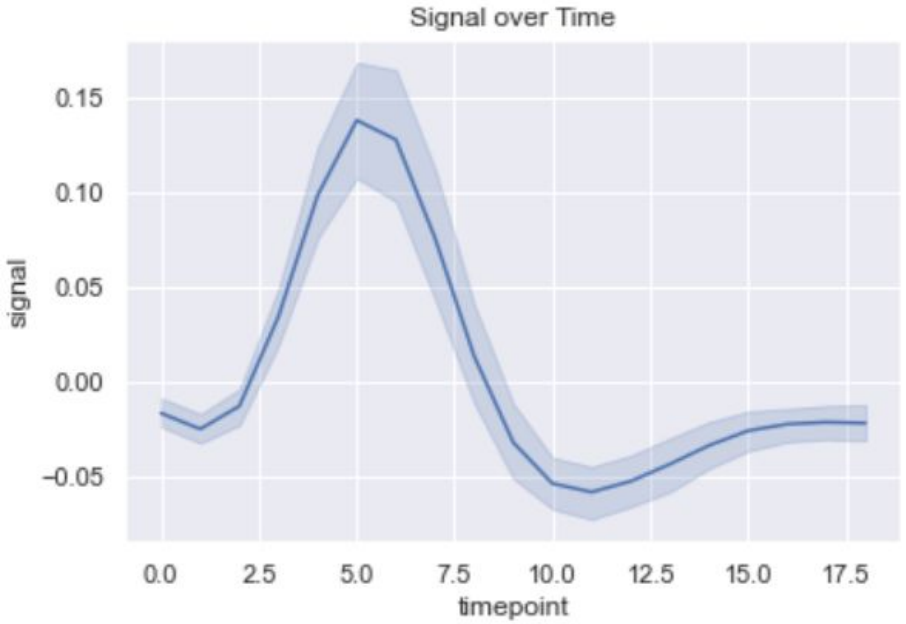




2 Numerical Variables: Line Plot

While scatter plots are useful in the general case when plotting two numerical variables, line plots are more appropriate when one variable does not or rarely repeat, such as in time series data.

The line plot on the right shows signal over time in FMRI time series data.



Additional Notes

- Some visualizations such as scatter plots can include a third variable by using a 3D visualization instead. However, 3D visualizations require a computer to view and sacrifice some readability.
- Numerical visualizations like scatter plots can also incorporate a categorical variable by using different colors (eg. blue points indicate category 1, orange points indicate category 2, etc.).

Normalizing and Standardizing Data

Motivation

Normalization and standardization are techniques commonly used to handle data where features have vastly different ranges of values. Seaborn's mpg dataset shown below is a good candidate for these techniques:

horsepower	weight	acceleration
130.0	3504	12.0
165.0	3693	11.5
150.0	3436	11.0
150.0	3433	12.0
140.0	3449	10.5

Standardization is also **required** before PCA.

When to Use Standardization and Normalization

- The decision to standardize and normalize an input is dependent on the kind of algorithm that you wish to use
- In general, algorithms that needs a similarity metric between samples are sensitive to transformations whereas others might be invariant
- These algorithms include SVMs, regularized Regression, k-means clustering and NNs
- Stochastic Gradient descent is also sensitive to feature scaling, standardizing is recommended (learning rate is dependent on the feature scaling)

Normalization

Normalizing a feature maps its values to the range [0, 1]. For a data point x_i, the following equation performs the normalization:

$$x_{i,normalized} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

In the next slide, we show an example of how normalization fixes the range of two different features, so that both features have values of the same magnitude.

Normalization Example

This slide shows how the “horsepower” and “weight” features of the mpg dataset are normalized. Notice that even though the features have different min and max values, they both end up with 0 and 1 as the new min and max after normalization.

```
mpg["horsepower"].describe()
```

```
count    392.000000
mean     104.469388
std       38.491160
min       46.000000
25%       75.000000
50%       93.500000
75%      126.000000
max      230.000000
```

```
mpg["horsepower_normalized"].describe()
```

```
count    392.000000
mean       0.317768
std        0.209191
min        0.000000
25%        0.157609
50%        0.258152
75%        0.434783
max        1.000000
```

Horsepower

```
mpg["weight"].describe()
```

```
count     398.000000
mean     2970.424623
std       846.841774
min      1613.000000
25%      2223.750000
50%      2803.500000
75%      3608.000000
max      5140.000000
```

```
mpg["weight_normalized"].describe()
```

```
count    398.000000
mean       0.384867
std        0.240103
min        0.000000
25%        0.173164
50%        0.337539
75%        0.565637
max        1.000000
```

Weight

Standardization

While standardization sounds similar to normalization, it does something fundamentally different. While normalization fixes the minimum and maximum values of the feature, standardization fixes the mean and standard deviation of the feature to be 0 and 1 respectively. Given a data point x_i , we have the following equation:

$$x_{i,standardized} = \frac{x_i - x_{mean}}{x_{std}}$$

This shows how much each data point varies from each other. In the next slide, we show how standardization applies to the same features.

Standardization Example

This slide shows how the “horsepower” and “weight” features of the mpg dataset are standardized. Notice that the new mean is roughly 0 (numerical imprecision) and the new standard deviation is 1. Additionally, the maximum and minimum describe how much the furthest points vary from the mean.

```
mpg["horsepower"].describe()
```

```
count    392.000000
mean     104.469388
std       38.491160
min       46.000000
25%       75.000000
50%       93.500000
75%      126.000000
max      230.000000
```

```
mpg["horsepower_standardized"].describe()
```

```
count    3.920000e+02
mean    -3.231542e-16
std      1.000000e+00
min     -1.519034e+00
25%     -7.656144e-01
50%     -2.849846e-01
75%      5.593651e-01
max      3.261284e+00
```

Horsepower

```
mpg["weight"].describe()
```

```
count     398.000000
mean     2970.424623
std       846.841774
min      1613.000000
25%      2223.750000
50%      2803.500000
75%      3608.000000
max      5140.000000
```

```
mpg["weight_standardized"].describe()
```

```
count    3.980000e+02
mean    -9.637740e-17
std      1.000000e+00
min     -1.602926e+00
25%     -8.817168e-01
50%     -1.971143e-01
75%      7.528861e-01
max      2.561961e+00
```

Weight

Additional Notes

- You cannot both standardize and normalize a feature; this is because normalizing transforms the feature's range to [0, 1], so the new mean of the feature will be larger than 0 (except in the degenerate case when all values are the same in a feature). This means you must pick one or the other.
- There exists a version of standardization that is more robust to outliers by using median and interquartile range instead of mean and standard deviation. You can read more about sklearn's robust scaler here.

One-Hot Encoding Categorical Data

Motivation

One-hot encoding transforms categorical variables into a more useable format. Trying to fit a model on a categorical column with string labels is difficult, since a computer is unable to process these strings the same way they do numbers. Additionally, while a string label might contain meaningful information to humans, a computer does not discern any such information.

This section discusses how one-hot encoding handles categorical variables.

Naive Approach: Enumeration

One naive approach to handling categorical variables is to use enumeration. Given that a column contains a set of string labels {label 1, label 2, ..., label n}, one could simply replace them with corresponding numbers {1, 2, ..., n}.

In the Titanic dataset, we see this in action with the passenger class column:

String labels:

class
Third
First
Third
First
Third

Enumerated labels:

pclass
3
1
3
1
3

Problems with Enumeration

The main problem with enumeration is that enumeration adds a mathematical constraints that weren't originally present. Mapping "First Class" to 1 and "Third Class" to 3 in the Titanic dataset adds nonsensical information that 3 * "First Class" is equal to "Third Class", or "Second Class" is equal to "Third Class" + 1.

These relationships between passenger classes obviously don't exist in the real world, so we need to find another way to handle this categorical data.

One-Hot Encoding Technique

The main idea behind one-hot encoding is to map each label to a list of 0-1 indicators, where label i corresponds to the ith indicator being activated. The length of the list is equal to the number of unique labels. We then have the following mapping:

Label 1: [1, 0,..., 0]

Label 2: [0, 1, 0, ... , 0]

Label n: [0, ..., 0, 1]

By removing the original categorical column and adding the list as new columns, we can successfully transform our data.

One-Hot Encoding on the Titanic Dataset

Applying this encoding to the passenger class column of the Titanic dataset, we get the following transformation:

String labels:

class
Third
First
Third
First
Third

One-hot labels:

class_First	class_Second	class_Third
0	0	1
1	0	0
0	0	1
1	0	0
0	0	1

We then add remove the “class” column from our data and add the columns “class_First”, “class_Second”, and “class_Third” to complete our transformation.

Additional Notes

- Not all “numerical” data is quantitative. In addition to the previously mentioned enumeration, where a number serves as a proxy for a class, data such as zip codes or phone numbers where mathematical relationships are unclear can also be encoded.
- If the number of unique labels is very large, one-hot encoding may produce a transformed dataset that has unreasonably many columns. This means that you may need to perform additional preprocessing the categorical column first, such as grouping rare labels together as “Other”.

Processing Text Fields with Regular Expressions

Motivation

Uncleaned text fields are commonly found in real world datasets, especially in data collected from surveys. Being able to discern meaningful information from strings that may be poorly formatted or contain unwanted punctuation and text is a vital skill needed to work with these datasets. Some fields such as natural language processing (NLP) also require cleaned text data.

This section discusses the fundamentals of regular expressions (regex), which can help process these fields.

What is Regex?

Regex is a useful programming tool that allows you to extract features from text, replace substrings, and perform other string manipulations. Regex allows you to create a set of characters, or a **pattern**, that can match substrings in other input strings.

Patterns are more useful than looking up a specific set substring in an input string, since they are general. Multiple substrings can potentially match a single pattern, as long as they follow the pattern's format.

Common Regex Functions

Below is a list of commonly used regex functions. Note that regex is imported as “re” in Python:

- *re.search(pattern, text)*: Detects whether regular expression pattern is present in the given text.
- *re.sub(pattern, substitute, input)*: Substitutes substrings in input text that match the given pattern with the substitute substring.
- *re.findall(pattern, string)*: Returns a list of all pattern matches in a string.
- *re.compile(pattern)*: Stores the regex pattern in cache memory for faster searches, which is useful if a pattern is reused many times.

Literals

Next, we can begin to discuss regex patterns, starting with literals. Literals are just strings with no catches, and match exactly with text that's identical with itself when included in a pattern.

For example, the regex pattern "https://eecs.berkeley.edu/" will match exactly the string "https://eecs.berkeley.edu/", and will fully match with no other string.

However, partial matches can be found as well, such as in "https://eecs.berkeley.edu/news", where the domain substring matches the pattern.

Character Sets and Meta Sequences

Character sets match any character that are contained in the set; they are denoted by square brackets. Listed below are commonly used ones:

- [xyz]: Matches exactly the characters “x”, “y”, or “z”.
- [a-z]: Matches any lowercase alphabetic character.
- [A-Z]: Matches any uppercase alphabetic character.
- [a-zA-Z]: Matches any alphabetic character.
- [0-9]: Matches any digit between 0 and 9.

These can also be inverted by using the ^ operator. For example, [^0-9] matches all non-numeric characters. Commonly used sets may also have a corresponding meta sequence; [0-9] is also represented by \d.

Quantifiers and Wildcards

In addition to the previously mentioned character sets and meta sequences, a period (.) is a wildcard matches any single character.

Character sets, meta sequences, and wildcards only match single characters. Quantifiers can specify how many times they need to match:

- *: The preceding character needs to match 0 or more times
- +: The preceding character needs to match 1 or more times
- ?: The preceding character matches 0 or 1 times
- {m, n}: The preceding character matches between m and n times

Basic Regex Example

Task: Extracting a float value from an uncleaned dollar string (eg. “$1,200,689.84 USD").

```
float(re.sub(r"[^\d.]", "", "$1,200,689.84 USD"))
```

```
1200689.84
```

The pattern first finds all characters that are **not** (^) numeric (\d) or periods (.), and replaces them with empty strings. This removes the dollar sign, commas, and suffix of the string, creating a valid string that can be converted into a float.

Additional Notes

- If you want to specify a special character such as a plus sign or period as a literal that's not in a character set, you need to escape the character with backslash.
- The pipe operator (|) specifies an or, which may be useful for specifying different possible characters/substrings at a location in the pattern.
- A great site to debug or sanity check your regex is https://regex101.com/, which highlights exactly what parts a pattern matches in a given input string and pattern, as well as explaining why the match exists.

Handling Imbalanced Class Distributions

Motivation

When performing classification on real world data, one problem that may arise is having an imbalance in the class distribution of your dataset. In an extreme case, suppose 99% of your dataset is class A and the remaining 1% of your dataset is class B; a naive classifier would be able to attain a good training accuracy of 99% by simply classifying everything as class A! However, this is clearly not an intelligent classifier.

This section discusses how we can modify our dataset to overcome this challenge.

Alternatives to Dataset Modification

While the focus of this project is on data cleaning, this slide briefly discusses alternatives to solving class imbalance besides preprocessing the dataset.

1. Collecting more data: While this solution may seem obvious, getting more data is one of the best ways to overcome class imbalance. However, this is not always possible, since data collection may be difficult or expensive.
2. Use a different evaluation metric: Our naive classifier always outputs the majority class since our accuracy metric penalizes false positives and false negatives equally. If you want your classifier to be more sensitive in one direction, consider using a different metric. A receiver operating characteristic (ROC) curve can help you determine how sensitive your model is in both directions.

Resampling Datasets

In addition to the methods proposed in the previous slides, another way to combat class imbalance is to resample your dataset. There are a few ways to perform resampling:

1. Undersampling: Sample a subset of data points from the majority class, so that fewer majority class points are included.
2. Oversampling: Sample data points from the minority class with replacement, so that minority class points have greater representation.
3. Generating synthetic data: Generate new synthetic data for the minority class; the SMOTE method is commonly used. Synthetic data is similar in concept to oversampling, except data points are not exact copies.

A Warning about Resampling

The primary concern about using resampling to handle class imbalance is that it shows an incorrect distribution of classes to the model during training. In our earlier example, if the population truly were comprised of 99% class A and 1% class B, then our dataset accurately reflects these proportions. By resampling our dataset, we are presenting our classifier with inaccurate proportions.

As a result, if you need to carefully consider if changing the reality of the dataset is appropriate before using a resampling method.

Filling Null/Missing Values

Motivation

Missing or null values are common problems that plague real world datasets. Whether they arise from errors that are later removed from the dataset, budget constraints that limit data collection, or another source, missing/null values need to be handled before we can train a model on a dataset.

This section discusses how to preprocess our data accordingly to handle null and missing values.

Two Approaches to Handling Null Values

While there are a multitude of ways to handle null values depending on if they occur systematically or at random, what kind of model is being trained on the data, and if the variable is categorical or numerical, these methods all fall under two main, overarching techniques:

1. Deletion: Remove the offending variables or rows that contain null values.
2. Imputation: Substitute the null values with some kind of filler value.

Method 1: Deletion

There are two simple ways to delete null values; you can delete all rows that contain null values, or you can drop the variable that contains null values.

1. Deleting rows: Deleting rows is appropriate when there's only a small number of null values, and **if the data is missing at random**. If there is a pattern where the data is null, deleting these rows may lose valuable information and produce biased models.
2. Dropping variables: Dropping a variable is appropriate when a large portion of the variable's values are null (eg. optional field in a survey with few responses), and the variable is insignificant.

Method 2: Imputation

There are multiple ways to impute null values.

1. Imputation with summary statistic: You can replace all null values in a variable with the variable's mean, median, or mode. Mean and median apply to numerical data while mode applies to categorical or discrete numerical data. While these substitutions are fast, they may be prone to bias and take no advantage of relationships between other variables.
2. Linear Regression: You can use a correlation matrix to select good predictors of the variable to impute and train a linear regression model to predict the null values. While this works in theory, the model may overfit and also makes the strong assumption that a linear relationship exists.

Method 2: Imputation (cont.)

3. K-Nearest Neighbors: Similarly to linear regression, a kNN model is trained on the dataset, where the imputed variable are what the model predicts. A distance metric between data points is decided, and the model uses the present values of the nearest neighbors to impute the null value.

While other methods exist in addition to the three listed, they are more complex and require prerequisite knowledge that’s out of scope for this project.

Additional Notes

- For categorical variables, it may also be appropriate to simply treat null values as a separate category instead of imputing or deleting.
- With all techniques listed, it is important to consider if the data is missing at random or systematically. Systematically missing data is much more difficult to deal with, since it may be indicative of some unseen, underlying pattern.

Removing Outliers (without OMP)

Motivation

Like missing values, outliers are another problem that are rarely encountered in classroom settings, but frequently plague real world datasets. Outliers are rare points that are either far above or far below the majority of the data; these points can skew the results of models trained on the data.

This section discusses some of the basic ways to identify and deal with outliers.

Defining Outliers

The main challenge with removing outliers is that there is always some ambiguity on whether or not a specific data point is actually an outlier. There exist many different definitions of this term. Here, we list three of the more common ways to discern if a specific data point is an outlier:

1. Data point violates constraints defined by domain knowledge of dataset
2. Data point is visually separated from other points on a graph
3. Data point fails a statistical test or rule

Method 1: Data Point Violates “Common Sense”

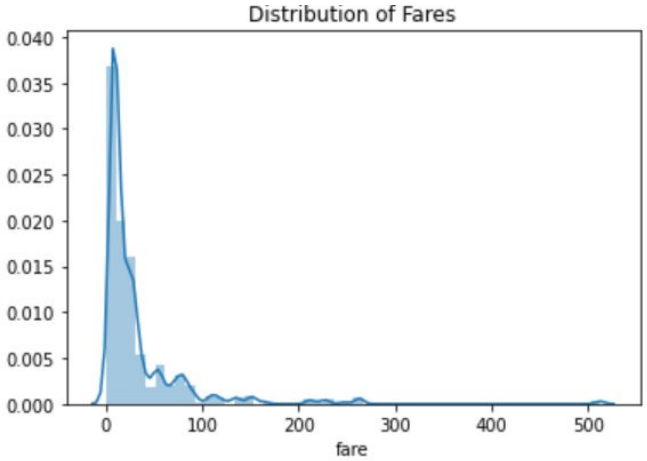
This first method involves identifying data points that violate “common sense” or domain knowledge assumptions about the dataset.

For example, if you are working with survey data and looking at the participant’s age, a specific value such as 280 would be considered an outlier, since you know that no person is 280 years old.

However, while it was clear in the example above that the data point is an outlier, that is not always the case. As a result, there is a subjective bias that can possibly be introduced into the dataset; make careful decisions based on your data when using this method.

Method 2: Data Point is Visually Separated on Plot

The second method involves plotting the data and deciding if specific points should be removed or not. The below plot shows the distribution of passenger fares in the Titanic dataset.



Like the first method, this method also introduces subjective bias since it forces you to make a “best judgement” call. While you could reasonably assume the points above 500 are outliers, what about the points between 200 and 300? This ambiguity makes this method challenging and inconsistent.

Method 3: Using Statistical Tests

The last method involves using statistical tests to determine if a specific data point is an outlier. There exist a plethora of different tests such as Peirce's Criterion, Chauvenet's Criterion, IQR rule, etc. Shown below is the IQR rule applied to Titanic passenger fares:

```
accepted_range = (titanic["fare"].mean() - 1.5 * IQR, titanic["fare"].mean() + 1.5 * IQR)
accepted_range
```

```
(-68.4301920314254, 132.8386079685746)
```

The IQR rule states that any fare lower than the mean - 1.5 * IQR or larger than mean + 1.5 * IQR is an outlier. While these tests are more objective than the previous methods, you still need to choose the appropriate one to use.

Additional Notes

- It is not always advantageous to remove outliers. If a work is very critical, such as studying the relationship between temperatures and airplane engine failures, removing outliers would be inappropriate.
- Make sure that outliers removed are sparse; if there are many outliers with similar values, that may hint at some underlying true pattern in the data.
- In addition to the methods we've listed, there exists another way to determine outliers with Orthogonal Matching Pursuit (OMP) that is out of scope for this project.

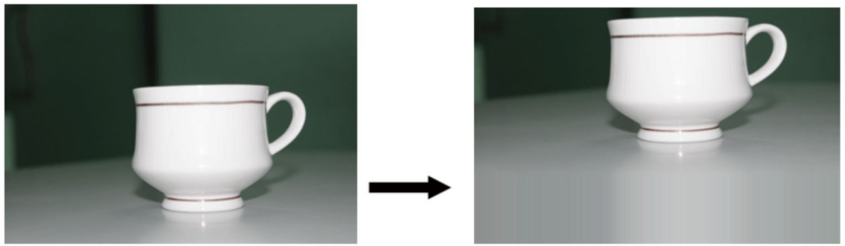
Image Data Augmentation

Motivation

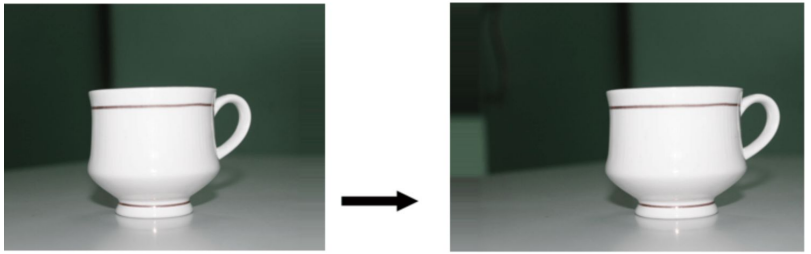
Image data augmentation refers to the techniques used to increase the effective size of our training dataset by introducing variations of existing samples. This is important because a larger training dataset means our trained model becomes more robust to the test and deploy environment, as we have artificially enhanced the comprehensiveness of the test dataset

Horizontal and Vertical Shifting

Shifting involves moving the focus of the image up or down, left or right by some number of pixels.



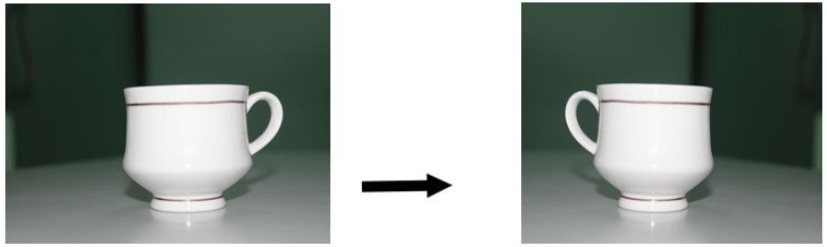
Ex. Vertical shift



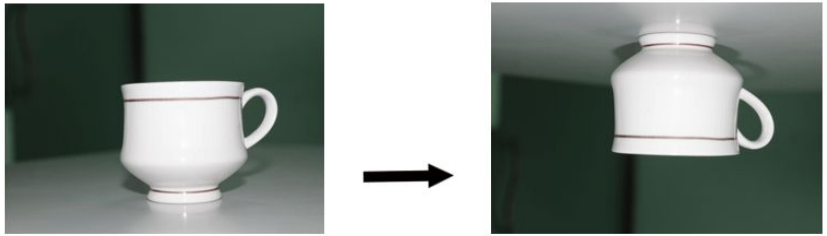
Ex. Horizontal shift

Horizontal and Vertical Flipping

Flipping reflects the image across either axis.



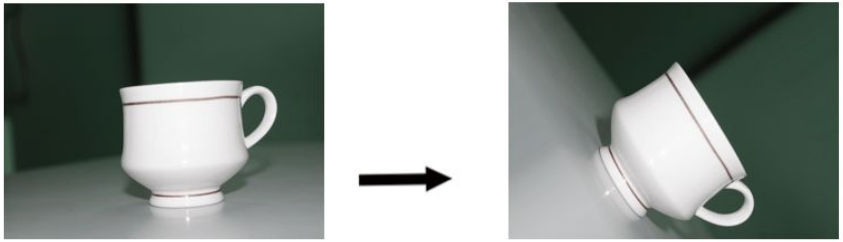
Ex. Horizontal flip



Ex. Vertical flip

Random Rotating

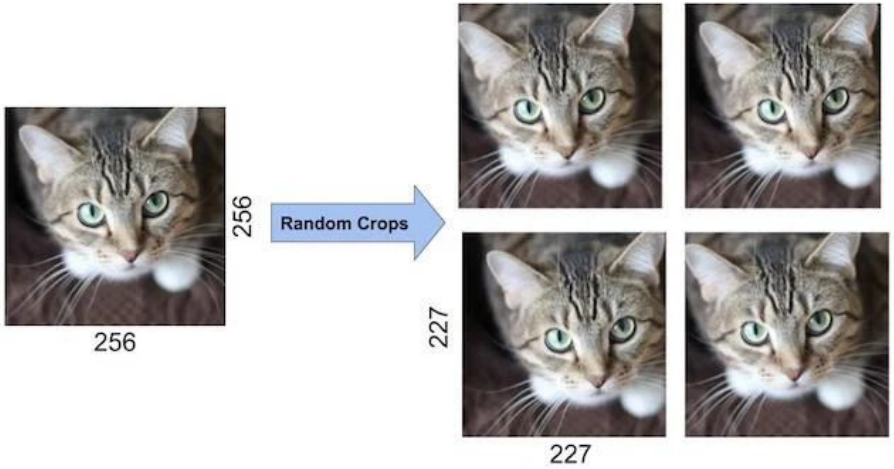
Rotation involves rotating the image at a random angle



Ex. Random rotation

Random Cropping

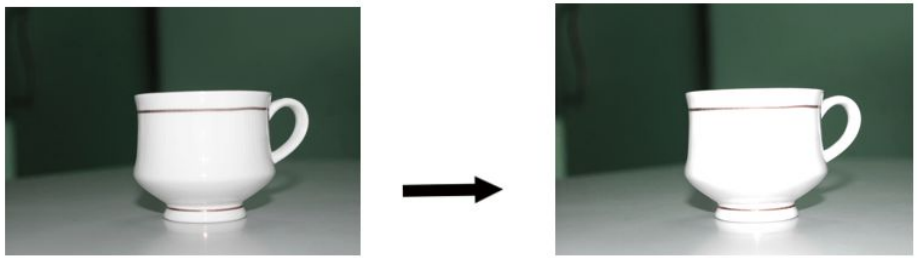
Cropping involves taking a smaller random crop of the original image.



Ex. Random cropping

Random Brightness

Random brightness involves altering the intensities of the RGB channels in the training images. The exact algorithm involves PCA and is described in the Alexnet paper (2012).



Ex. Random brightness

Standard practices (Common transforms)

Standard practice for image data augmentation is as follows:

1. Scale resolution down to 256 x 256
2. Take a random 224 x 224 crop (alternatively, consistently take crops of corners and center)
3. Generate horizontal flip
4. Perform brightness augmentation (optional)

Standard practices (How it is applied)

Augmentation is usually performed at the dataloader step.

1. Training pipeline requests a batch of images during an epoch
2. Dataloader grabs a batch of raw images from the saved folder of training / val data
3. Dataloader applies transforms (perhaps with random factors) onto batch of raw images
4. Dataloader sends transformed images as input into model

Data Decorrelation and Whitening

Correlation in Data

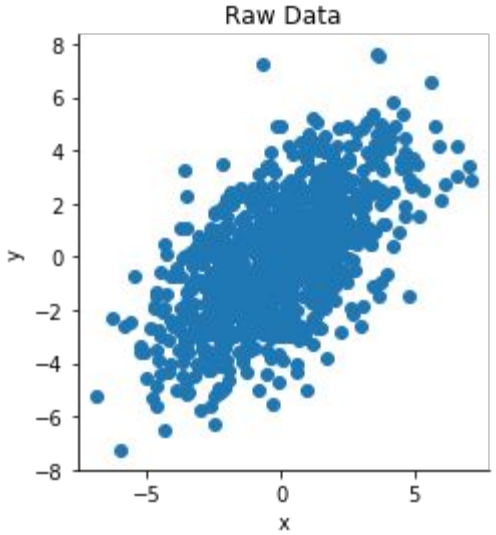
- The concept of a correlation matrix for our data matrix should be familiar since this was part of teaching SVD and PCA in EE16B
- In general, we would like for our data to have uncorrelated features:
  - Uncorrelated features makes the optimization of Neural Networks more efficient
  - Correlations in data potentially means that there is redundant information which means that we can compress our data
- Our discussion of decorrelation and later whitening will depend heavily on the PCA/SVD topics discussed in class, so if you are unfamiliar, we recommend brushing up with those topics

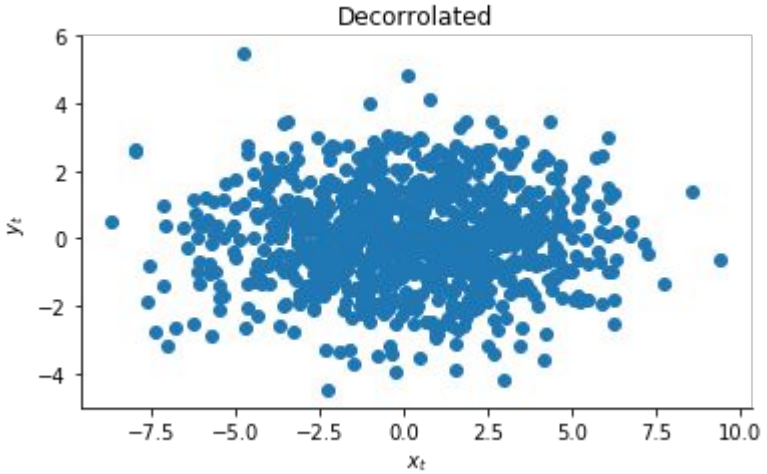
Decorrelating Data

- To decorrelate data, it means to remove the any correlations between different features in the data
- We decorrelate data by zero meaning the features and multiplying by the U matrix that we get from the SVD
- This allows us to get a data matrix that has a diagonal correlation

$$\tilde{X} = U^\top X \implies \tilde{X}^\top \tilde{X} = U^\top U \Lambda U^\top U = \Lambda$$

Decorrolated Data Example





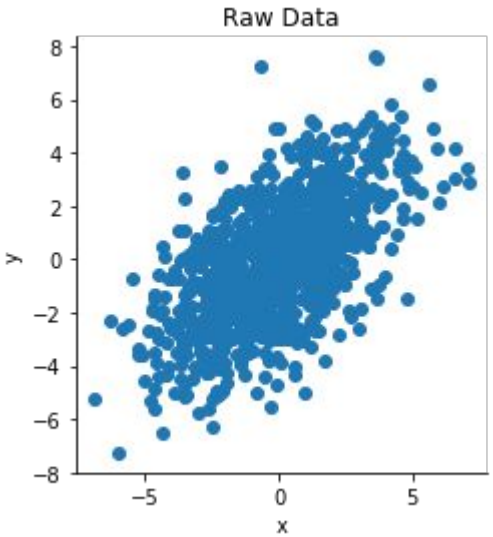
As we can see, after decorrelating the data, the features are now independent of each other

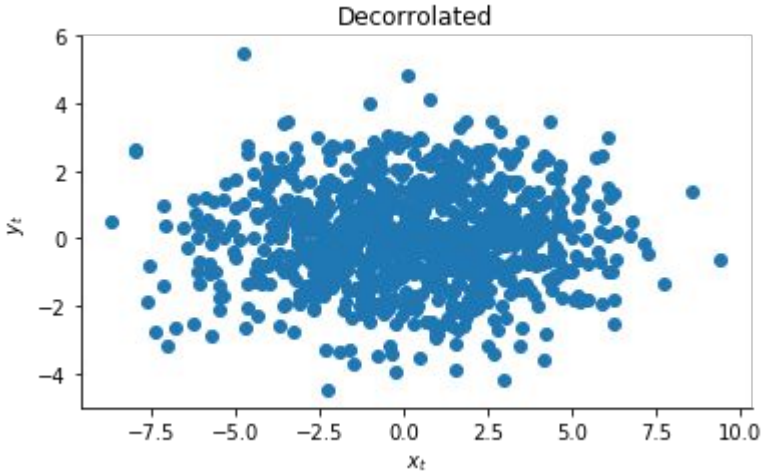
Whitening Data

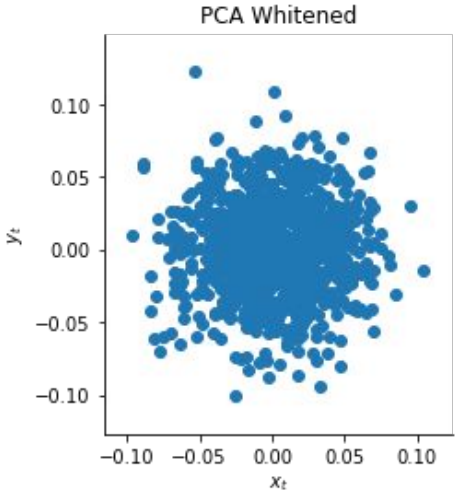
- Whitening is a linear transformation used to both decorrelate different features and set the variances of different features to be identical
  - This names comes from the term “white noise”
- In our case, it means that we wish for the covariance matrix of the data to be the identity matrix rather than just diagonal
  - We can accomplish this by simply scaling each feature by the singular values:
  - This also known as PCA Whitening

$$\tilde{X} = \Lambda^{-1/2} U^\top X \implies \tilde{X}^\top \tilde{X} = U^\top U \Lambda^{-1/2} \Lambda \Lambda^{-1/2} U^\top U = I$$

PCA Whitening example







As we can see, PCA whitening lets us transform the data into a spherical blob of data

ZCA Whitening

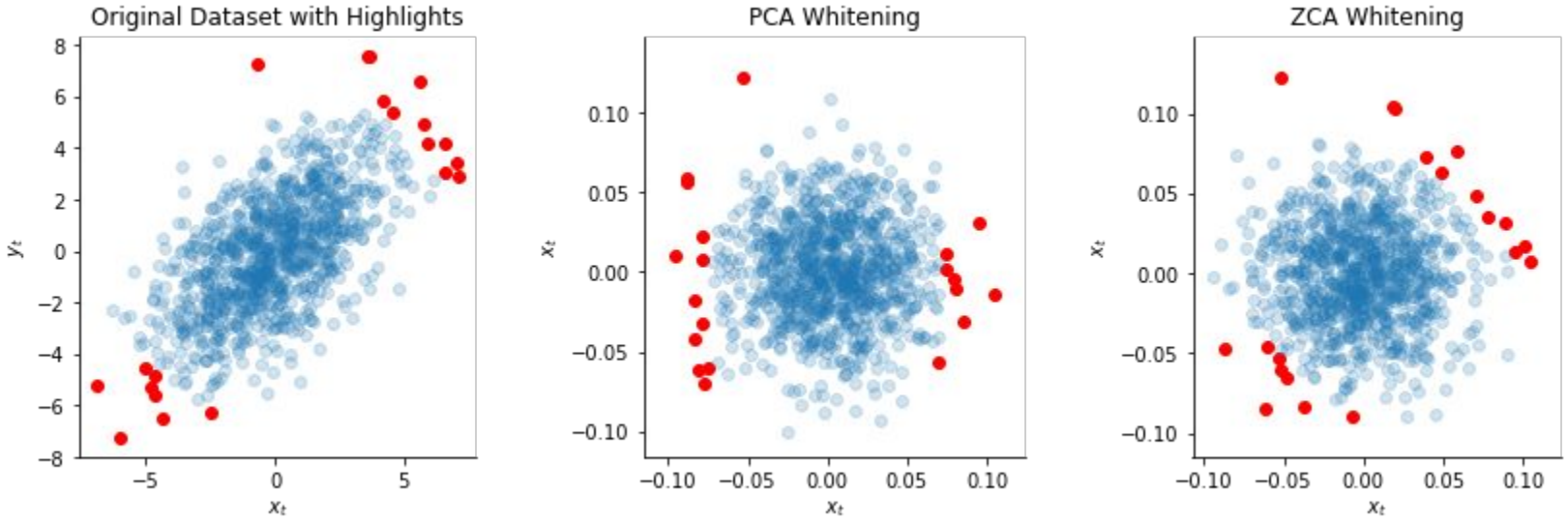
- One thing of note is that the linear transformation by PCA whitening is not unique: any rotation matrix applied on top of the rotation matrix maintain a identity as the covariance
- In Zero-Phase Component Analysis (ZCA) whitening, we decide to choose the matrix U as our rotation matrix:

$$\tilde{X} = U\Lambda^{-1/2}U^{\top}X$$

PCA whitening vs. ZCA whitening

- PCA whitening is ideal if your goal is to try and compress the data after applying whitening
- You can always just keep the top k principal components, and you will still have identity as your covariance
- ZCA whitening is ideal if you want the transformed random vector to be as similar as possible to the original vector
    - See source for the exact proof which we will leave out here

PCA vs ZCA Whitening



The orientation of the data is loosely preserved with ZCA

Remarks

- Avoid the common pitfall of not applying the transformations from the training set to the test set when testing
- The effect of whitening can be substantial on neural networks, but it really depends on the specific optimizer
  - Whitening is not commonly used in Convnets, standardizing and zero-mean is more common
  - See this source for more about gradient descent