Basic Data Cleaning Practices

Nicholas Lee* **Patrick Wang†** **Kevin Hsu‡** **Arjun Bharat§**
Department of Electrical Engineering and Computer Science
University of California at Berkeley
Team .N.A.N.T.S. Final Project T

1 Data Cleaning and Why We Do It

Data Cleaning is the act of preprocessing and transforming raw data into a format that is easier to train and learn on. Several techniques that you have seen before in EE16A and EE16B fall under the catagory of data cleaning, including, but not limited to outlier removal with OMP and PCA for dimensionality reduction. From market research and interviews, we have found that around 70% of active time and effort spent in industry is based around collection, preprocessing, and cleaning data for training. The reason for this is because lots of times, the quality of data used to train a model can make or break the performance of said model. High quality data can potentially make a very simple model work where as complex, low quality data could require much more complex and difficult models to perform. Even if a model is published online with the architecture and hyperparameters, training it on less ideal datasets can lead to a drastic drop in performance. Perhaps if we had an extreme quantity of data, we could potentially learn from the raw data; however, this is not very likely in practice and basic data cleaning and preprocessing can significantly improve the performance of a machine learning algorithm when data is not necessarily abundant.

1.1 Making Data Legible

At the most basic level, data cleaning is about making data legible. When we perform machine learning, especially in a classroom context, we often believe that our data is legible to the algorithm. However, this is not usually the case in reality. Datasets in the real world are messy. When humans have to submit the data or machines mindlessly scrape the data from the internet, we have lots of missing or malformed data. Null values, inconsistent labels, spelling errors are often rampant. Data that doesn't make sense such as negative values for age or fractional values for discrete fields can be found in your dataset; sometimes you might see a 200 year old man. Other times, you might see inconsistent data, where the same feature maps to two different labels or duplicate entries in the table. In any case, data cleaning should resolve these issues so that the data is coherent; having a legible dataset is a necessary prerequisite to training any machine learning model. Sometimes, these errors can only be found through carefully constructed screens.

2 Data Cleaning Techniques

2.1 Data Visualization [Optional Review Topic]

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

*caldragon18456@berkeley.edu
†pwang2000@berkeley.edu
‡kevin.hsu@berkeley.edu
§arjunb@berkeley.edu

Preprint. Under review.

explain our findings to others that might want to learn about our work. Since this topic should have already been covered in another project, this section has been marked as optional review.

Before listing what plots this section will cover, here are some basic rules you would always want to follow:

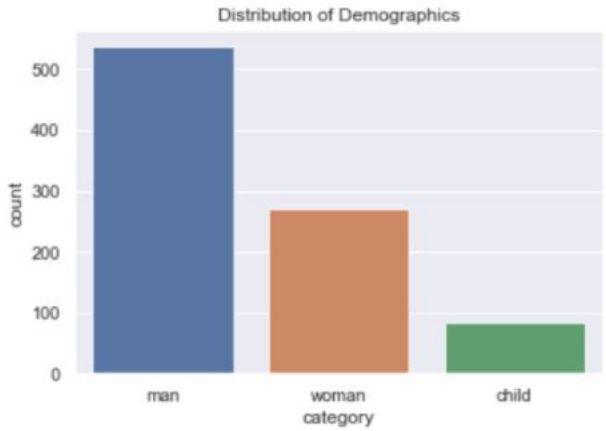
1. Add descriptive titles and axis labels.
2. Use appropriate scaling so that your data is readable.
3. Avoid plotting too much data and cluttering your visualization (overplotting).
4. Consider if your data is categorical or quantitative before choosing a plotting technique.

This list of rules is by no means a comprehensive list, but they are good guidelines to keep in mind. 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. These rules help ensure that your visualizations achieve these goals.

This rest of this section will describe four basic plots:

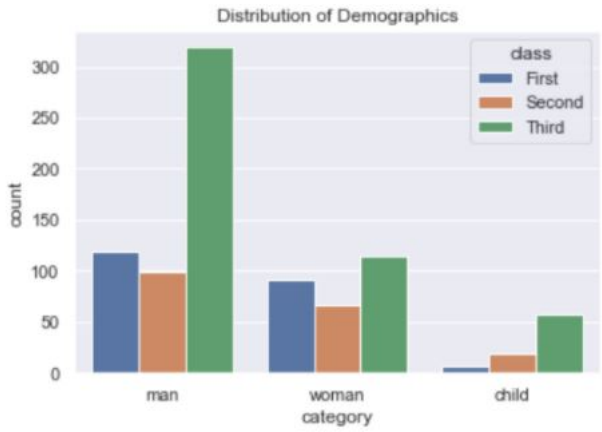
1. Bar Charts
2. Multi-level Bar Charts
3. Histograms
4. Scatter Plots

The bar chart describes a single categorical variable. Shown below is an example of a bar chart for the demographics of passengers of the Titanic:



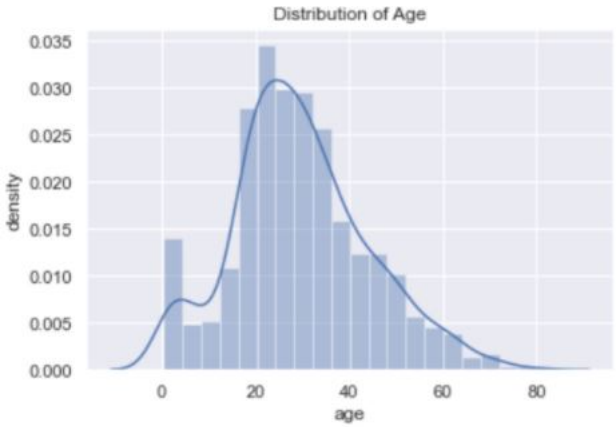
In a vertical bar chart, the x axis displays all unique values the categorical variable can take, and the y axis shows their counts or densities. This allows a bar chart to concisely describe how often each label appears in the variable.

A multi-level bar chart is similar to a bar chart, but it can display 2 or more categorical variables at once. Shown below is a 2-level bar chart for the demographics and class of Titanic passengers:



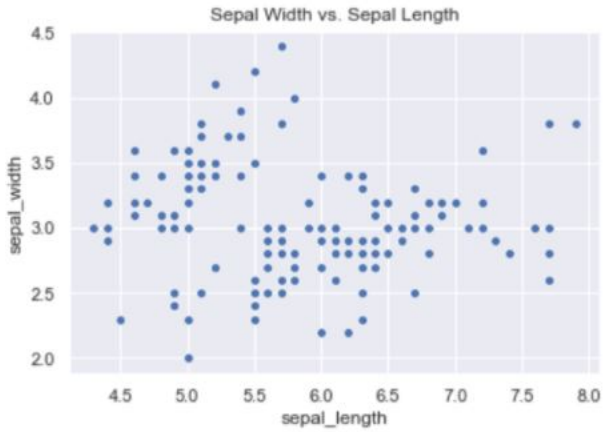
This graphic concisely displays the distribution across both the class and demographics categorical variables. While it is possible to display even more variables, multi-level bar charts with more than 2 levels are rarely used, as that could clutter the visualization and hinder the effectiveness of the plot.

The histogram describes the distribution of a single quantitative variable. Shown below is an example of a histogram of ages for Titanic passengers:



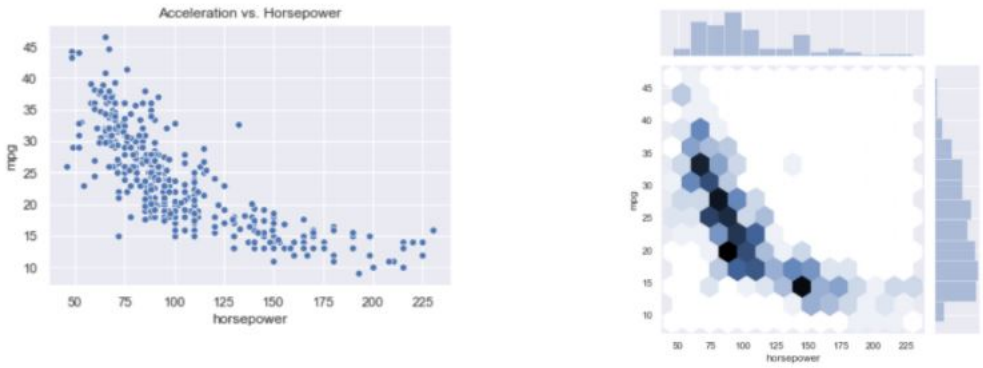
In a histogram, the x axis displays the range of values the quantitative variable can take, and the y axis shows their counts or densities. This parallels the bar chart, but a range of values is placed on the x axis instead of the unique labels a bar chart uses to describe categorical variables.

Finally, a scatter plot describes how two (or three) quantitative variables are related to each other. Shown below is a scatter plot of how sepal width and sepal lengths are related in the Iris dataset.

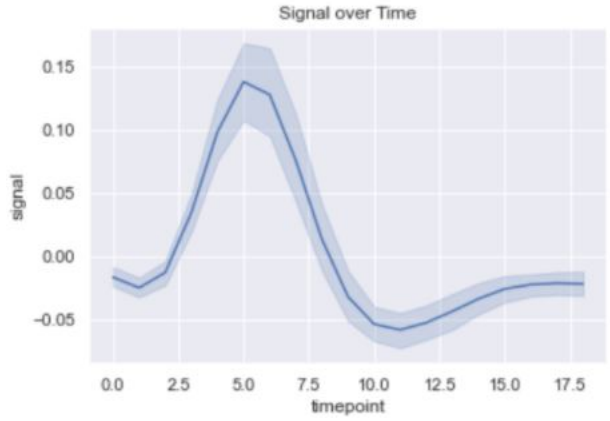


The axes of this plot each describe a variable. This implies that to plot three categorical variables, we would have to use a 3D scatter plot. 3D plots are only situationally useful, as they require a computer to view and are more difficult to read compared to a 2D scatter plot. Scatter plots are useful for showing the pattern between variables in a concise fashion.

However, while scatter plots are powerful visualizations, there are notable variants of scatter plots that are situationally better. For example, overplotting can diminish the readability of scatter plots. Thankfully, a variant of scatter plots called hex plots can help with this issue by interpolating regions of points together. Shown below is an example an overplotted scatter plot and its corresponding scatter plot.



The hex plot more accurately portrays what's exactly going on in the cloud of points in the scatter plot. Another useful variant of scatter plots are line plots. If one variable is sparse or does not repeat, such as in time series data, line plots can show a more clear pattern than scatter plots by connecting the individual points together. Shown below is a line plot drawn for time series data:



2.2 Normalizing and standardizing numerical data

In general, it is a good idea to normalize and/or standardize data before training on it. **Normalization** is the practice of rescaling a specific feature in a dataset to lie between 0 and 1. In order to normalize a feature in a dataset, we apply the following transformation to the column:

$$\tilde{x} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{1}$$

Applying normalization across each numerical feature in a dataset allow us to get a dataset with a common (unitless) scale on each of the features without changing the relative difference between different data points. **Standardization** is the practice of rescaling a feature so that it has a mean of zero and a standard deviation of 1. We apply this transformation by applying the following transformation:

$$\tilde{x} = \frac{x - \mu}{\sigma} \tag{2}$$

where the mean $\mu = \frac{1}{n}\sum_i x_i$ and the standard deviation $\sigma = \sqrt{\frac{1}{n}\sum_i (x_i - \mu)^2}$

There are plenty of different reasons why we should normalize or standardize data.

For example, consider a regression problem, where one wishes to predict both the cost and the size of a home based on features like the income of the people living inside of it, the number of people, and the average income of the area. We can see here that the number of people living the house will likely be orders of magnitude smaller than all of the other features. In addition, the size of a home and the cost that it sells for can also vary by orders of magnitude. Thus, we could potentially see that the algorithm is more accurate for the cost of the home compared to the size because the average cost of a home is much larger than the size of a home.

On the other hand, we would also see that one of the features in our inputs would be much smaller than the others. For example consider trying to use a SVM with a two dimensional feature space. If the range of one feature space is much larger than another, then we can see that one of the dimensions becomes much more important than the other, which would skew results.

The question is when should we normalize/ standardize our data at all. In general, we would do these transformations as a good rule of thumb if we have no domain knowledge of the problem. This standardized procedure makes exploring the data and analysis easier. However, if we have domain knowledge of the problem and the algorithm we will use to training the data on, then we will have to make choices. In general, any algorithm that is not scale invariance will require normalization or standardization. Usually, these are algorithms that depend on a similarity metric between data points such as the Euclidian distance. You can see a nice chart of the algorithms that require it here: [12]. There are also other small caviots to this statement. For example, while Linear Regression is not scale invariant (the weight vector can scale with what the scaling is), regularized linear regression is not. In particular, the regularization constants are supposed to depend on the optimal weight vector, which changes due to the scalings. On the other hand, while a fully connected layer in a neural network is invariant to affine transformations, the optimization algorithm is not in general. At a high level, we can only apply one learning rate across the entire gradient as we run backpropogation. Thus, if different features may require different learning rates based on the scalings. You can see a more advanced discussion of this topic here: [11] Other unsupervised clustering algorithms like k-means are affected by these transformations. An extended discussion of this topic can be found here: [8]

2.3 One-hot encoding categorical data

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, a string label might contain meaningful information to humans in the form of context clues or domain knowledge. For example, if you are studying housing prices, you may be able to ballpark or a price for a building based on just a city label due to your domain knowledge. In contrast, a computer does not discern any such information. This section discusses how one-hot encoding handles this problem of categorical data.

However, before discussing one-hot encoding, let us try a more naive approach to handling categorical variables: enumeration. Given that a column contains unique columns {label 1, label 2,... label n}, we can substitute these with the corresponding numbers {1, 2,..., n}. This means if a certain data point x is label i for $i \in \{1, 2, ...n\}$, then x will be substituted with i after enumeration. While enumeration does transform the categorical variable into a numerical variable that can be passed into a model, it is also problematic because it introduces mathematical constraints that weren't present in the original dataset. For example, suppose again you were working with housing data. Your categorical variable is the house's city, which can be one of the three following choices: Berkeley, Oakland, San Francisco. After enumeration, Berkeley maps to 1, Oakland maps to 2, and San Francisco maps to 3. However, this mapping introduces nonsensical relationships into the dataset, such as 3 * Berkeley = San Francisco, or San Francisco - Oakland = Berkeley. These relationships obviously don't exist in the real world, so we need a different solution.

One-hot encoding fixes this by using indicators. Given that your categorical variable has n unique labels, one-hot encoding creates n indicator variables. For a given data point, the ith indicator signals if that data point is label i, and all other indicators will be 0. As a visual example, in the previous scenario with housing data, if we let labels 1, 2, and 3 be Berkeley, Oakland, and San Francisco respectively, a data point that's labeled as Oakland will correspond to a one-hot encoding

of $\{0, 1, 0\}$. This is because indicator 2 for Oakland is on, but the indicators 1 and 3 for Berkeley and San Francisco are off. This encoding is better than enumeration because it does not introduce any nonsensical relationships.

Two concerns to be wary of for one-hot encoding are numerical yet categorical data and categorical variables with many unique values. Some variables can be numerical yet still categorical, and thus require one-hot encoding; zip codes are such an example, as they are more similar to other categorical information like city or county than they are to any quantitative variable. Additionally, using one-hot encoding on a categorical variable with many unique values can lead to a giant dataset, since the number of unique values is the number of new variables added. To amend this, some additional preprocessing might need to be done, such as grouping less common labels into a designated "Other" label.

2.4 Preprocessing text fields with regular expressions

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.

To process these text fields, we will be using regular expressions (regex). Regex is a powerful programming tool that allows you to extract features from text, replace substrings, and perform other string manipulations. This is accomplished by creating a set of characters, or a pattern, that can match substrings in input strings. Patterns are more useful than just directly looking up a substring from input text, because any substring that matches the pattern's format will be matched. The next paragraphs describe how to create a pattern, as well as what it means for a string to match a pattern.

However, before discussing regex patterns, we provide a short list of commonly used regex functions below. Note that regex is imported as re in Python.

1. re.search(pattern, text): Detects whether a regex pattern matches any substring in the given text.
2. re.sub(pattern, substitute, text): Replaces all matches of the pattern within the text with substitute. This can be used to remove all matches by setting substitute to an empty string.
3. re.findall(pattern, text): Finds all substrings where the pattern matches substrings in the text.
4. re.compile(pattern): Stores the regex pattern in cache memory. Useful when you want to reuse a pattern many times.

Additionally, the syntax for applying regex to pandas series differs slightly from using re directly. However, understanding these basic functions will allow you to pick up the new syntax quickly.

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.

To improve on just using literal patterns, we can introduce character sets and meta sequences. Character sets are denoted by square brackets [], and match any character that are contained within the set. Listed below are some commonly used character sets:

1. $[abc]$: Matches exactly with the characters "a", "b", or "c".
2. $[a - z]$: Matches any lowercase alphabetic character.
3. $[A - Z]$: Matches any uppercase character.
4. $[a - zA - Z]$: Matches any alphabetic character.
5. $[0 - 9]$: Matches any digit between 0 and 9.
6. $[8@a - z]$: Example of mixing; matches the number 8, an @ symbol, and any alphabetic characters.

These sets can also be inverted with the $\wedge$ operator. For example, $[\wedge 0-9]$ matches with any character that's not a number. Additionally, many commonly used sets also have meta sequences that can be used as a shorthand. For numeric set $[0-9]$, \d is the corresponding meta sequence that can be used as substitute.

In addition to character sets, the period (.) is a wildcard matches ANY character.

So far, with just literals, character sets, and meta sequences, we can write patterns that match sequences of fixed length. How can we write patterns that match with variable length text? This is where quantifiers become helpful. Quantifiers follow a literal, meta character, character set, or wildcard and specify how many times that character can be repeated. Listed below are a few useful quantifiers:

1. *: The preceding character repeats 0 or more times.
2. +: The preceding character repeats 1 or more times.
3. ?: The preceding character matches 0 or 1 times.
4. {m, n}: The preceding character matches between m and n times.

Finally, now that we've learned about literals, quantifiers, and character sets, we can start writing some regex statements. Let's walk through an example of a regex problem. Suppose that we have an uncleaned dollar string (eg. "$1,200,689.84 USD"), and we want to extract a float from this string. We can write the following line of Python code to do so:

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

```
1200689.84
```

The regex function chosen is re.sub, so the code will substitute any match in the input string with an empty string (specified in second argument), effectively removing it. As a result, we want our matches to be all characters that are not numeric characters or periods. \d matches all numbers while the period in a character set is treated as a literal instead of a wildcard, so it will match a literal period. The $\wedge$ negates the character set, so it will match characters that are NOT numeric or periods. This means that the dollar sign, trailing space, and "USD" will match. This will return a new string "1200689.84" that can be converted into a float value.

This regex guide is by no means comprehensive, but it at least builds the foundations required to learn more regex. Additionally, a great resource to use when you are learning regex is https://regex101.com/, which allows you to input your own patterns and strings and highlights exactly what parts a pattern matches, as well as explaining why that match exists.

2.5 Handling imbalanced class distributions

When we do classification, one thing we never get into is the actual distribution of classes within the dataset. When we work with nice standard datasets like MNIST and Cifar10, we get to work with balanced class distributions, where we have approximately the same amount of samples per class. However, in the field, we may not so fortunate. In many cases, we may see or even expect to see there to be an inbalance in the distribution. For example, consider the classic problem of trying to predict if a person has a rare disease. If less than 5% of the population has this disease, we may expect to see that the amount of people without this rare disease will severely outnumber the people with the disease.

This brings us to the paradox of accuracy. Consider a binary classification problem where only 1% of the dataset is of one class, the vast majority are in another class. Naively, we would be able to just classify every single sample as one class and get a 99% accuracy trivially, without any learning. If this is also true of the test set and of real data points that you see in the future at deployment, you may always hover around this 99% accuracy, which doesn't seem useful. This is because usually, in these cases, we would want to be able to have good accuracy on multiple classes rather than just 1 class. While this is an extreme case, there are also other cases in which this may cause problems.

Usually the case is that we don't have enough data for the minority class, so generative algorithms such k-means may struggle to properly identify the minority class.

In this assignment, we will deal with directly augmenting or adjusting the dataset to try and rebalance the dataset. For example, in the case of image data, we could potentially add more augmented images to the dataset (detailed in section 3) in order to oversample the minority class. In other cases, we may have enough data from the minority class, but an extreme amount of samples from the minority class. In this case, we may do undersampling and specifically choose to randomly leave out samples from the majority class in order to get a balanced class distribution

One popular method of augmenting a dataset to balance the classes is the Synthetic Minority Oversampling Technique (SMOTE). [1] With SMOTE, we select members of the minority class and try to synthesize results between different examples in the feature space. The steps are as follows:

1. Select a random example from the minority class
2. Pick the k nearest neighbors of that example (k is a hyperparameter)
3. Pick a random neighbor from the k nearest neighbors
4. Generate a new sample using the original example and its neighbor

Generating the new sample can be done in different ways. Usually, we take a convex hull of the two samples to get a synthetic example. For example, given two samples $x_1, x_2 \in \mathbb{R}^n$, we would construct a synthetic example with:

$$x_{syn} = ax_1 + (1-a)x_2,\ 0 < a < 1 \tag{3}$$

One thing to note is that this approach only makes sense in a case where features have no spacial correlation i.e. non-image data. In addition, in cases where there is significant overlap between different classes, this may introduce more noise than signal.

One thing of note is that sometimes balancing classes is not necessary depending on the situation. For example, you may want this imbalance to be a part of your algorithm. In the case where you are trying to do something like anomaly detection, where a false positive could potentially be costly, you might want to have your algorithm largely classify every example as negative. (Future assignments will cast light on this topic.) Other times, we can use different

2.6 Handling null/missing values

Missing or null values are a challenging problems that are rarely encountered in classroom settings, but frequently 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.

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

Deletion is the simpler of the two techniques. You can either choose to delete any rows (or data points) or delete the columns (features) that contain null values. Deleting rows is only appropriate when there's a small number of null values, and if the data is missing at random. If there is a pattern in where the data is null, deleting these rows may lose valuable information and produce biased models. Unfortunately, this assumption is rather difficult to enforce. Dropping columns or features is appropriate when a large portion of the values are null (for example, data collected from an optional field with few responses in a survey), and if the feature itself is not influential in training the model.

On the other hand, there are many different ways to impute null values. Imputation is the act of replacing null values with some type of substitute. A very simple imputation method is to use a summary statistic; you could simply take the mean, median, or mode of all non-null values for that feature and fill in all nulls with that statistic instead. Note that for categorical variables, the only summary statistic that can be used is mode, since you can't take the mean or median of a categorical variable. Additionally, another option for categorical variables is to not substitute or delete! You could simply treat null values as another category. While these summary statistics are fast and easy, they may be prone to bias and don't take advantage of relationships between other variables to fill in the

null values. The second imputation method is to use a linear regression model to predict the feature with null values. You can use a correlation matrix to find other features that are good predictors of the feature you are trying to impute. Then, you can train a linear model and use it to predict the null values. While this approach sounds good in theory, it often performs worse in practice, as the linear model can overfit easily and forces a strong assumption that a linear relationship exists. The last imputation method this note will discuss is to use a k nearest neighbor (kNN) approach. The basic premise of the kNN approach is similar to that of linear regression: train a model on the other features to predict the null value. You first define a distance metric between the data points; then use the distance metric to find the k nearest data points where the value for the feature is not null. Finally, use a summary statistic on the neighbors to calculate a value that can be filled for the null value. Beyond these methods, there exist many more techniques to impute null values.

Dealing with null values is a complicated process that can potentially require involved algorithms. In order to choose the appropriate method to successfully preprocess these null values, you must always consider factors such as how the data is missing, what kind of model you're training, and if the variable is quantitative or qualitative.

2.7 Removing outliers (without OMP)

Like missing/null values, outliers are also problems common problems real world datasets that are only infrequently seen in classroom settings. Outliers are rare points that lie far above or below the majority of the data; outliers are problematic because they are not necessarily representative of the underlying patterns that the majority of the data is drawn from, so training a model on outliers may cause the model to perform erratically. As a result, we may want to remove these outliers from our data before training our model. Note that we may not always want to remove outliers, especially in cases where our model must be very sensitive. For example, if we are studying data that describe what range of temperatures an airplane engine fails in, removing outliers may not be appropriate.

Given that removing outliers is appropriate, we then encounter our main challenge: defining what constitutes as an outlier. There are many different definitions on what makes a point an outlier, which leads to ambiguity in determining if a specific data point should be removed or not. Algorithms like orthogonal matching pursuit (OMP) exist to help with this task of removing outliers are out of scope for this project. Instead, we offer three methods to determine if a data point should be removed:

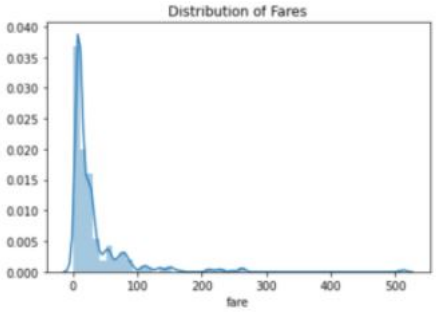
1. Data point violates constraints defined by domain knowledge of the dataset.

2. Data point is visually separated from other points when plotted on a graph.

3. Data point fails a statistical test or rule.

The first method is to identify outliers by finding points that violate our preconceptions about the dataset, or, in other words, violate "common sense". For example, suppose you are working with survey data, and one of the fields is the participant's age; you would have preconceived notions about what the distribution of values look like. If you then see a specific value of 200, you know that this data point is completely unreasonable in that it violates common sense. However, the main issue with this method is that it introduces a subjective element into your dataset. While it is immediately evident in the exaggerated example that a 200 year old participant is an outlier, what if someone lists 100 as their age? While this might be highly unlikely, depending on what type of survey is being conducted, it is definitely still possible. Making a best judgement call on these types of points is often difficult and may lead to inconsistent decisions; as a result, one must be especially careful when using this method.

The second method is to plot your numerical data on a graph and identify outliers by looking at which points are visually separated from the remainders. After using a visualization like a histogram to plot the distribution of a feature, you can pick out which points are further away from the majority of the data. For example, below is an image of passenger fares from the Titanic dataset.



However, like the first method, this method must be used with caution as well, since it also introduces a subjective element into the outlier removal process. For example, while the few points greater than 500 can be safely classified as outliers, what about the points between 200 and 300? Deeming a point to be "far away enough" from the remainder of the data by just looking at it is not a rigorous process, and the ambiguity involved can again lead to inconsistent behavior.

Finally, the last method is to use a statistical rule or test. There exist a plethora of different tests and rules available; just to list a few, some commonly used tests include Peirce's criterion, Chauvanet's criterion, and the IQR rule. Unfortunately, there's no single test that is guaranteed to always give the best result. This again implies that a best judgement call is required to pick an appropriate test. Still, this offers more rigor given that an appropriate test was chosen.

The main takeaway from this section is that outliers are difficult to contend with, since these methods all rely on some degree of human judgement. It is up to the individual performing the data cleaning to carefully consider their options and rely on their experience and reasoning to decide on a good solution.

2.8 Data decorrolation and whitening

In EE16B, you should have learned about the SVD and PCA from the prospective of analyzing correlations and the sample covariance in the form of $\frac{1}{n}A^\top A$ (or $\frac{1}{n}AA^\top$, depending on the conventions that you use for data in columns or rows). If you are unfamiliar with these topics, we would recommend you brush up on these concepts.

An important step in data preprocessing is decorrolation: applying an affine transformation to the data that gives us features that are uncorrelated: $\frac{1}{n}A^\top A = \mathbb{I}$. The practice of decorrelating your data has some key benefits:

- Decorrelating your data can potentially reduce the variance in your data. If I have multiple features that are correlated, we can consider decorrolating data as an averaging transformation, where we take the average of the features. This reduces noise because the average of a noisy measurement decreases its variance. At the most basic case, consider a feature vector of size m where every single feature is a feature $x_i = x + \epsilon_i$ with some independent noise. You can do the maths, but we expect that after running PCA, we would be able to get an average value for x, which reduces the noise in this feature.
- In particular for algorithms that depend on gradient descent, decorrolating the inputs can improve convergence via the condition number of the Hessian, which is beyond the scope of this course. [17] This idea is similar to the idea behind why normalization/ standardization works for neural networks: optimization algorithms are not invariant to linear transformation and these preprocessing steps can help convergence of gradient based algorithms

The first step of decorrelating is always making your features zero-mean. Then ,we take this zero mean matrix and leverage the SVD. Consider that the covariance matrix is equal to:

$$AA^\top = U\Lambda U^\top \tag{4}$$

Consider that since the matrix U is orthogonal, we can simply apply the transformation $\tilde{x} = U^\top x$ to get a data matrix where the features are now uncorrelated. To understand this transformation, notice that the matrix U represents the column span of the dataset. Thus, by applying inverse of the matrix to the matrix, we are effectively transforming our data to the feature space span by the columns of

A. Since the columns of A are linearly independent, we have transformed our data to a vector space where the features are uncorrelated.

Whitening is a linear transformation where the data is transformed to a feature space where features have no correlation and identical variance. The name whitening comes from the idea of "white noise" that has mean zero and variance 1. Again, we can leverage the SVD in order to accomplish this. Consider the squared singular values $\Lambda = diag(\sigma^2)$. Let us define the "inverse square root" of this matrix as $\Lambda^{-1/2} = diag(1/\sigma)$. This matrix is also diagonal and notice that it has the property that $\Lambda^{-1/2}\Lambda\Lambda^{-1/2} = \mathbb{I}$. Thus, we should define the transformation

$$x_{pca} = \Lambda^{-1/2}U^\top x \tag{5}$$

which we will call the PCA whitening transformation. Notice that this has the effect that

$$\tilde{A}\tilde{A}^\top = \Lambda^{-1/2}U^\top U\Lambda U^\top U\Lambda^{-1/2} = \mathbb{I} \tag{6}$$

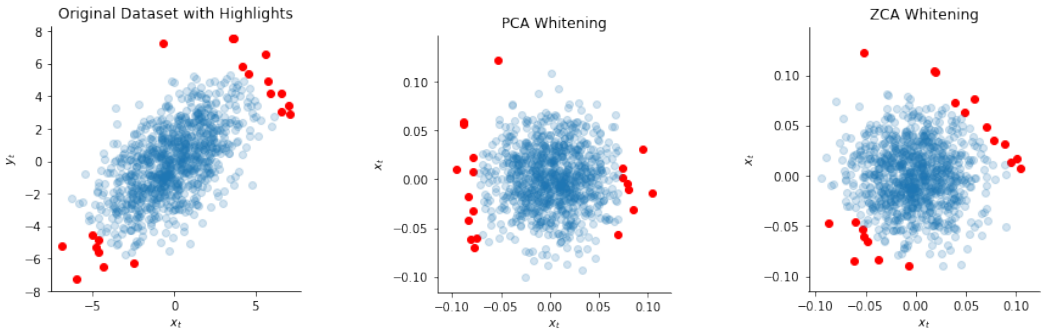
The way that this transformation works should not be surprising if you understand the SVD and PCA. Once we have transformed our dataset to the feature space of the column space, we are now seeing that these features have non-uniform variances. These variances are exactly defined by the singlar values of the matrix. Thus, by selectively multiplying the new features by the inverse of the square root of the variance, we can now have a new feature that has unit variance.

Notice now that any PCA whitening matrix is not unique. In fact, any rotation matrix that satisfies the relation $R^\top R = RR^\top = \mathbb{I}$ can be multiplied to the transformation to get a new dataset $x_{rotated} = R\Lambda^{-1/2}U^\top x$ which also has identical variances and independent features. Now, we may want to choose a rotation matrix to apply to the matrix. (If we choose not to, we are implicitly choosing the identity as our rotation matrix.) Well, in Zero-Phase Component Analysis (ZCA) whitening, we just choose the column span matrix U as our roation matrix, making our transformation

$$x_{zca} = U\Lambda^{-1/2}U^\top x \tag{7}$$

The question is why would we choose these two transformations in general.

- The PCA whitening transformation is useful if you wish to compress the data after whitening the dataset. For instance, instead of multiplying by the full matrix $U^\top$, we could implicitly choose just K of the first columns. This has the effect of choosing the features that already had the highest variance as our features before normalizing the features.
- The ZCA whitening transformation is useful if your goal is for the transformed matrix to be as close to the original data as possible. Indeed, it can be shown that the ZCA whitening procedure minimizes the Frobinious norm of the difference between the original and whitened data, which is out of scope of these notes. [9]



In this case, I have highlighted some of the extremities of the dataset. As you can see, PCA whitening implicitly means that you are rotating the data in such a way as to align the data vectors along the principle components. However, if you wish for the data to have approximately the same orientation after whitening, ZCA will rotate the data back into the original feature space.

At the end of the day, there is also the question of whether you want to whiten your data at all rather than just standardizing or normalizing the data. This operation requires the use of a covariance matrix

and the SVD, which are infeasible if you have very large dimensions and lots of data points. In addition, the variability of the dataset is potentially something that you desire because it gives you information about the task at hand. If you have outside domain knowledge of the task at hand, you can make educated choices on how to preprocess your data, but if you have no idea, whitening can have a positive effect on your algorithm's success.

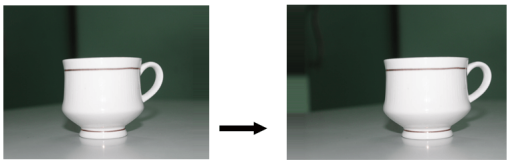
3 Image Preprocessing and Augmentation

3.1 Overview

Image data augmentation refers to the set of techniques used to increase the effective size of the training dataset. This is done by generating variations of the existing samples during training. Common image augmentations include horizontal and vertical shifting, random rotations, random cropping, and other more exotic augmentations such as brightness shifts and random blurs. The payoff for using data augmentation is that we have a more complex and varied training dataset, so the learned model is more robust to variation and less prone to overfitting and saturating performance on the training dataset.

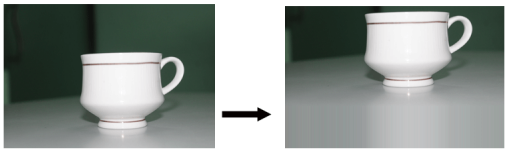
3.2 Examples

Horizontal shifting example:



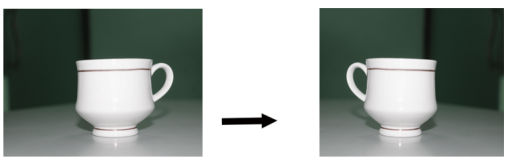
Here, the image is shifted a couple of pixels to the right. This is done by moving the pixels to the right and adding dummy pixels to the left.

Vertical shifting example:



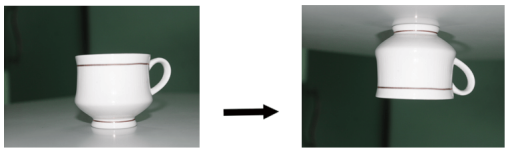
Here, the image is shifted a couple of pixels up. This is done by moving the pixels upwards and adding dummy pixels to the bottom.

Horizontal flip example:



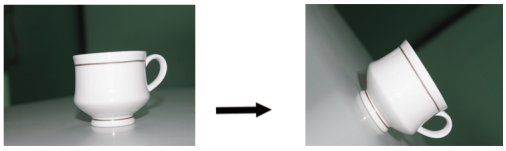
Here, the image is flipped horizontally. This is done by reversing the rows of the image.

Vertical flip example:



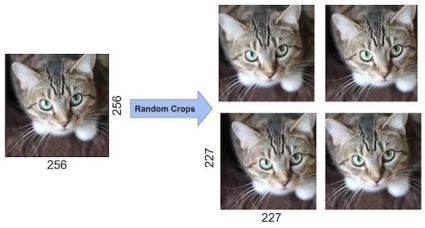
Here, the image is flipped vertically. This is done by reversing the columns of the image.

Random rotation example:



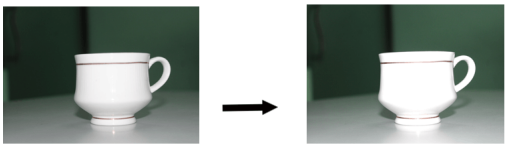
Here, the image is rotated a randomly picked number of degrees to the right. This is done through geometry, as the new location of each pixel can be calculated exactly. Some bound checking will have to be done as some pixels will be out of bounds in the final rotated image.

Random crop example:



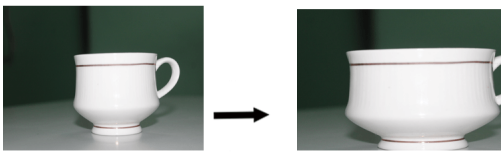
Here, the image is randomly cropped from the original image. This can be done multiple ways. If the target size is smaller, we can simply pick random indices and take the crop within those bounds. If the target size is the same, we can pad the sides with a set number of layers of dummy pixels and then pick our random indices for boundaries.

Random brightness example:



Here, the brightness of the image has been increased a random amount. The naive way to do this is to pick a random value and increase the each pixel values by that amount. In the literature, there is a more complex way to do it involving PCA scaling and adding to each RGB channel by the eigenvalue and eigenvectors as detailed in the original Alexnet paper. The exact implementation details are out of scope.

Random zoom example:



Here, the image has been zoomed in by a randomly chosen scale. This is done through pixel replication or interpolation; the exact implementation details are out of scope.

3.3 Workflow and Standard Practices

In practice, the low level implementations of all of these functions are given either through image processing libraries such as OpenCV, or even directly within machine learning frameworks such as PyTorch or Tensorflow. These standardized implentations are optimized to take advantage of the parallelness so in practice, you will always be using those directly.

The standard practice for augmenting a dataset can vary from task to task. As a baseline example, for image classification on Imagenet, the standard practice is to randomly resize the image down to 256 x 256, take a 224 x 224 random crop, do a random horizontal flip, move it onto a tensor (scaling it from 0-255 to 0-1), and normalize it with the RGB channelwise mean and stddev.

Augmenting happens during training through the dataloader. Given a dataset stored in a saved folder directory, the dataloader will pull batches of raw images during each batch of each epoch during training, apply the given transforms to the raw images, and pass on the transformed images as input into the first layer of the given model to be trained. This means for small tasks, we can be bottlenecked by the data transforming and loading process, but it saves huge amounts of storage as we don't have to store the transformed images anywhere, as we can delete them after passing them through our model once.

4 References and Sources

[1] Jason Brownlee. *SMOTE for Imbalanced Classification with Python*. Aug. 2020. URL: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

[2] Clemens Brunner Neuroscience. *Whitening with PCA and ZCA*. Dec. 2018. URL: https://cbrnr.github.io/2018/12/17/whitening-pca-zca/.

[3] Franck Dernoncourt. *How and why do normalization and feature scaling work?* Nov. 2012. URL: https://stats.stackexchange.com/questions/41704/how-and-why-do-normalization-and-feature-scaling-work.

[4] Omar Elgabry. *The Ultimate Guide to Data Cleaning*. Mar. 2019. URL: https://towardsdatascience.com/the-ultimate-guide-to-data-cleaning-3969843991d4.

[5] Kayla Ferguson. *When Should You Delete Outliers from a Data Set?* Jan. 3. URL: https://humansofdata.atlan.com/2018/03/when-delete-outliers-dataset.

[6] Charter Global. *What is data augmentation in image processing?* Apr. 2020. URL: https://www.charterglobal.com/image-data-augmentation-and-ai/.

[7] Hadrienj. *Preprocessing for deep learning: from covariance matrix to image whitening*. Aug. 2018. URL: https://hadrienj.github.io/posts/Preprocessing-for-deep-learning/.

[8] Leonard Kaufman and Peter J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Wiley, 2005.

[9] Agnan Kessy, Alex Lewin, and Korbinian Strimmer. "Optimal Whitening and Decorrelation". In: *The American Statistician* 72.4 (2018), pp. 309–314. DOI: 10.1080/00031305.2016.1277159. eprint: https://doi.org/10.1080/00031305.2016.1277159. URL: https://doi.org/10.1080/00031305.2016.1277159.

[10] Swetha Lakshmanan. "How, When, and Why Should You Normalize/Standardize/Rescale Your Data?" In: *Towards AI* (May 2019). URL: https://towardsai.net/p/data-science/how-when-and-why-should-you-normalize-standardize-rescale-your-data-3f083def38ff.

[11] Yann A. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

[12] Kevin Markham. *Comparing supervised learning algorithms*. Feb. 2020. URL: https://www.dataschool.io/comparing-supervised-learning-algorithms/.

[13] Madeline McCombe. *Text Processing is Coming*. June 2019. URL: https://towardsdatascience.com/text-processing-is-coming-c13a0e2ee15c.

[14] Andrew Ng. *PCA Whitening*. URL: http://ufldl.stanford.edu/tutorial/unsupervised/PCAWhitening/.

[15] Deborah Nolan. *Textbook Entry from Data 100: Principles of Data Science*. URL: https://www.textbook.ds100.org/ch/06/viz_intro.html.

[16] Baptiste Rocca. *Handling Imbalanced Datasets in Machine Learning*. Jan. 2019. URL: https://towardsdatascience.com/handling-imbalanced-datasets-in-machine-learning-7a0e84220f28.

[17] Christie Smith. *Conditioning and Hessians in analytical and numerical optimization - Some illustrations*. Reserve Bank of New Zealand Discussion Paper Series DP2007/06. Reserve Bank of New Zealand, Mar. 2007. URL: https://ideas.repec.org/p/nzb/nzbdps/2007-06.html.

[18] Alvira Swalin. *How to Handle Missing Data*. Jan. 2018. URL: https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4.

[19] Great Learning Team. *AlexNet: The First CNN to win Image Net: What is AlexNet?* June 2020. URL: https://www.mygreatlearning.com/blog/alexnet-the-first-cnn-to-win-image-net/.