Building Data Science and Machine Learning Pipeline

- **Typical Data Science Work Flow**
- **Challenges in the Data Science Work Flow**
- **Exploratory Data Analysis**
- **Feature Engineering**
- **Model Building and Performance Evaluation**
- **Model Deployment to Production**

Data Science pipelines are step by step sequences of processing and analysis applied to data for a specific purpose within a business or group. They're useful in production projects, and if one expects to encounter the same type of business question in the future, so as to save on design time and coding. For instance, one could remove outliers, apply dimensionality reduction techniques, and then run the result through a random forest classifier to provide automatic classification on a particular dataset that is pulled every week.

Challenges Associated with Machine Learning/ Data Science Pipelines

In creating the data science/ machine learning pipelines, there are challenges that data scientists face, but the most prevalent ones fall into three categories: Data Quality, Data Reliability and Data Accessibility.

https://www.datanami.com/2018/09/05/how-to-build-a-better-machine-learning-pipeline/

Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data.

To get the best possible results from a predictive model, you need to get the most from the data for your algorithms to work with

How do you get the most out of your data for predictive modeling?

This is the problem that the process and practice of feature engineering solves.

Actually the success of all Machine Learning algorithms depends on how you present the data.

*The features you use influence more than everything else the result. No algorithm alone, to my knowledge, can supplement the information gain given by correct **feature engineering**. — Luca Massaron*

Importance of Feature Engineering

The features in your data directly influence the predictive models you use and the results you can achieve. The better the features that you prepare and choose, the better the results you will achieve.

The results you achieve are a factor of the model you choose, the data you have available and the features you prepared. Even your framing of the problem and objective measures you're using to estimate accuracy play a part. Your results are dependent on many inter-dependent properties. *You need great features that describe the structures inherent in your data.*

Better features means flexibility.

You can choose "the wrong models" (less than optimal) and still get good results. Most models can pick up on good structure in data. The flexibility of good features will allow you to use less complex models that are faster to run, easier to understand and easier to maintain. This is very desirable.

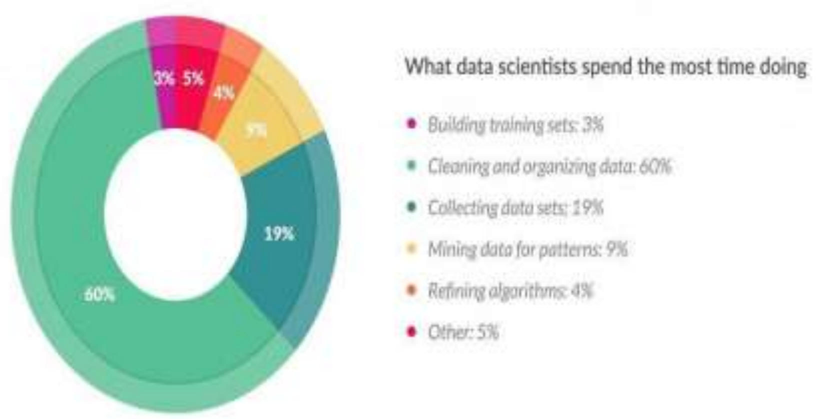
Better features means simpler models.

With well-engineered features, you can choose "the wrong parameters" (less than optimal) and still get good results, for much the same reasons. You do not need to work as hard to pick the right models and the most optimized parameters.

With good features, you are closer to the underlying problem and a representation of all the data you have available and could use to best characterize that underlying problem.

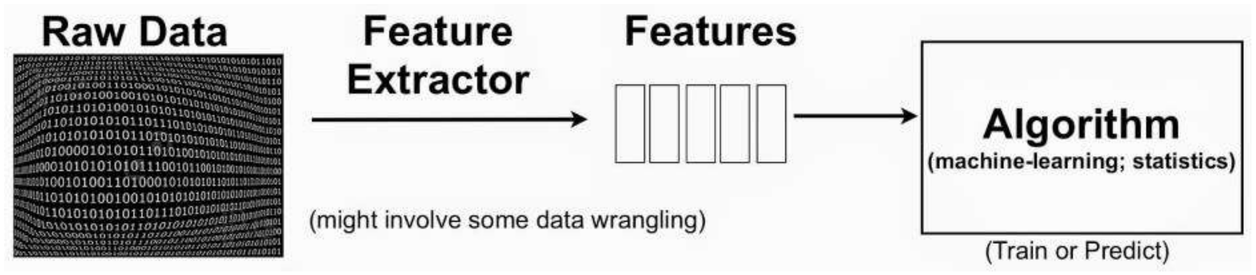
Better features means better results.
The algorithms we used are very standard for Kagglers. […] We spent most of our efforts in feature engineering. (from an Active Kaggler)

According to a survey in Forbes, data scientists spend **80%** of their time on **data preparation:**



Source: https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/

In this section, we will consider Supervised Learning and tabular data to look at the various Feature Construction Techniques that you like to might consider for your own problems


Raw Data
Feature Extractor
Features
Algorithm
(machine-learning; statistics)
(might involve some data wrangling)
(Train or Predict)

- Detecting and Treating Missing Values
- Detecting and Treating Outliers
- Binning (Bucketing)
- Log Transformation

- Grouping Operations
- Feature Splitting
- Decoding Date-Time Column
- Decoding Categorical Attributes
- Feature Scaling

1. Detecting and Treating Missing Values through Imputation

This is one of the most common problems you will encounter when you try to prepare your data for machine learning. The reason for the missing values might be human errors, interruptions in the data flow, privacy concerns, and so on. Whatever is the reason, missing values affects the performance of the machine learning models.

The simplest solution to the missing values is to drop the rows or the entire column. There is not an optimum threshold for dropping but you can use **70%** as an example value and try to drop the rows and columns which have missing values with higher than this threshold.

```
threshold = 0.7#Dropping columns with missing value rate higher than threshold
data = data[data.columns[data.isnull().mean() < threshold]]

#Dropping rows with missing value rate higher than threshold
data = data.loc[data.isnull().mean(axis=1) < threshold]
```

Numerical Variables Imputation

Imputation is a more preferable option rather than dropping because it preserves the data size. However, there is an important selection of what you impute to the missing values. I suggest beginning with considering a possible default value of missing values in the column. For example, if you have a column that only has **1** and **NA**, then it is likely that the **NA** rows correspond to **0**. For another example, if you have a column that shows the **"customer visit count in last month"**, the missing values might be replaced with **0** as long as you think it is a sensible solution.

Another reason for the missing values is joining tables with different sizes and in this case, imputing **0** might be reasonable as well.

Except for the case of having a default value for missing values, I think the best imputation way is to use the **medians** of the columns. As the averages of the columns are sensitive to the outlier values, while medians are more solid in this respect.

#Filling all missing values with 0
data = data.fillna(0)**#Filling missing values with medians of the columns**
data = data.fillna(data.median())

Categorical Variables Imputation

Replacing the missing values with the **maximum occurred value** in a column is a good option for handling categorical columns. But if you think the values in the column are distributed uniformly and there is not a dominant value, imputing a category like “**Other**” might be more sensible, because in such a case, your imputation is likely to converge a random selection.

#Max fill function for categorical columns
data['column_name'].fillna(data['column_name'].value_counts()
.idxmax(), inplace=True)

2. Detecting and Treatment of Outliers

The best way to detect the outliers is to demonstrate the data visually. All other statistical methodologies are open to making mistakes, whereas visualizing the outliers gives a chance to take a decision with high precision. Box plots, scatter plots and likes.

Statistical methodologies are less precise as I mentioned, but on the other hand, they have a superiority, they are fast. Here I will list two different ways of handling outliers. These will detect them using **standard deviation**, and **percentiles**.

Outlier Detection with Standard Deviation

If a value has a distance to the average higher than ***x * standard deviation,*** it can be assumed as an outlier. Then what **x** should be?

There is no trivial solution for x, but usually, a value between 2 and 4 seems practical.
#Dropping the outlier rows with standard deviation
factor = 3
upper_lim = data['column'].mean () + data['column'].std () * factor
lower_lim = data['column'].mean () - data['column'].std () * factor

data = data[(data['column'] < upper_lim) & (data['column'] > lower_lim)]

In addition, **z-score** can be used instead of the formula above. **Z-score** (or standard score) standardizes the distance between a value and the mean using the standard deviation.

Outlier Detection with Percentiles

Another mathematical method to detect outliers is to use percentiles. You can assume a certain percent of the value from the top or the bottom as an outlier. The key point is here to set the percentage value once again, and this depends on the distribution of your data as mentioned earlier.

Additionally, a common mistake is using the percentiles according to the range of the data. In other words, if your data ranges from **0** to **100**, your top **5%** is not the values between **96** and **100**. Top **5%** means here the values that are out of the **95th** percentile of data.

```
#Dropping the outlier rows with Percentiles
upper_lim = data['column'].quantile(.95)
lower_lim = data['column'].quantile(.05)

data = data[(data['column'] < upper_lim) & (data['column'] > lower_lim)]
```

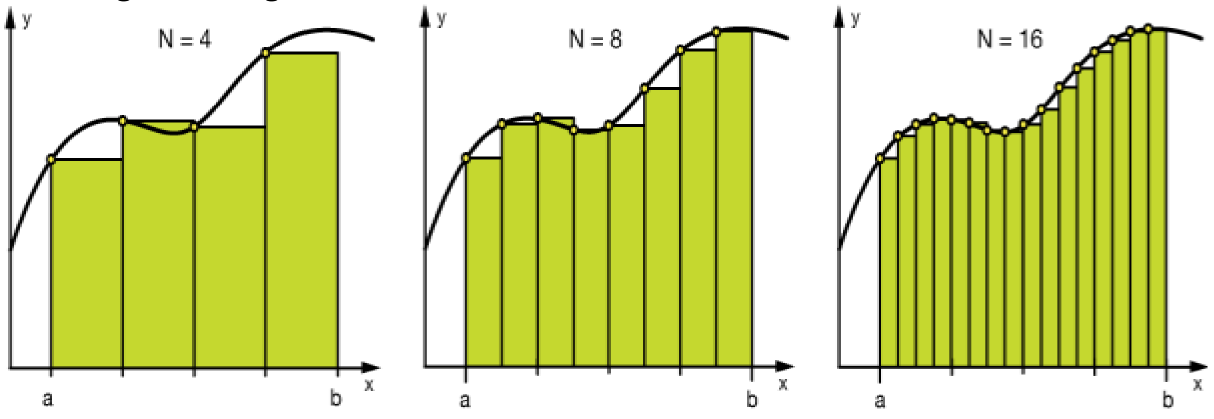
An Outlier Dilemma: Drop or Cap

Another option for handling outliers is to **cap** them instead of dropping. So you can keep your data size and at the end of the day, it might be better for the final model performance.

On the other hand, capping can affect the distribution of the data, thus it’s better not to exaggerate it.

```
#Capping the outlier rows with Percentiles
upper_lim = data['column'].quantile(.95)
lower_lim = data['column'].quantile(.05)data.loc[(df[column] > upper_lim),column] = upper_lim
data.loc[(df[column] < lower_lim),column] = lower_lim
```

3.Binning/ Bucketing



Binning illustration of numerical data

Binning can be applied on both categorical and numerical data:

Bucketing makes sense when the domain of your attribute can be divided into neat ranges, where all numbers falling in a range imply a common characteristic. It reduces overfitting in certain applications, where you don't want your model to try and distinguish between values that are too close by – for example, you could club together all latitude values that fall in a city, if your property of interest is a function of the city as a whole. Binning also reduces the effect of tiny errors, by 'rounding off' a given value to the nearest representative. Binning does not make sense if the number of your ranges is comparable to the total possible values, or if precision is very important to you.

The main motivation of binning is to make the model more **robust** and prevent **overfitting**, however, it has a cost to the performance. The trade-off between **performance** and **overfitting** is the key point of the binning process

4. Skewed and Wide Distributions Logarithm Transformations

It is one of the most commonly used mathematical transformations in feature engineering. What are the benefits of log transform:

- It helps to handle skewed data and after transformation, the distribution becomes more approximate to normal.
- In most of the cases the magnitude order of the data changes within the range of the data. For instance, the difference between ages **15** and **20** is not equal to the ages **65** and **70**. In terms of years, yes, they are identical, but for all other aspects, **5** years of difference in young ages mean a higher magnitude difference. This type of data comes from a multiplicative process and log transform normalizes the magnitude differences like that.
- It also decreases the effect of the outliers, due to the normalization of magnitude differences and the model become more robust.

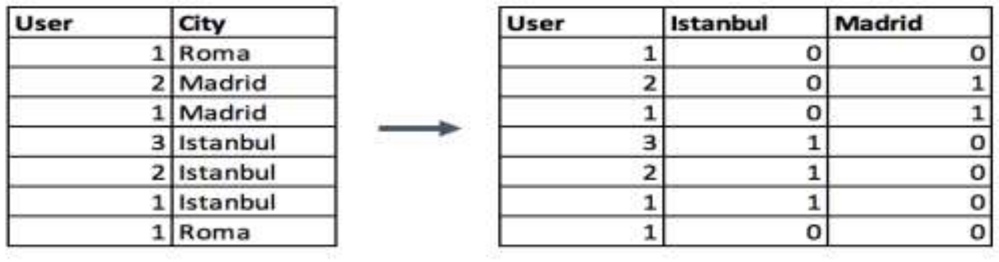
A critical note: *The data you apply log transform must have only positive values, otherwise you receive an error. Also, you can add* ***1*** *to your data before transform it. Thus, you ensure the output of the transformation to be positive.*

Symmetric distribution is preferred over skewed distribution as it is easier to interpret and generate inferences. Some modeling techniques requires normal distribution of variables. So, whenever we have a skewed distribution, we can use transformations which reduce skewness. For right skewed distribution, we take square / cube root or logarithm of variable and for left skewed, we take square / cube or exponential of variables.

- Cube root can be applied to negative values including zero. Square root can be applied to positive values including zero.

5. Decomposing Categorical Attributes

- Creating dummy variables: One of the most common application of dummy variable is to convert categorical variable into numerical variables. Dummy variables are also called Indicator Variables. It is useful to take categorical variable as a predictor in statistical models. Categorical variable can take values 0 and 1.
- Sklearn provides a very efficient tool for encoding the levels of a categorical features into numeric values. LabelEncoder encode labels with value between 0 and n_classes-1.
- One-Hot Encoding:One-Hot Encoding transforms each categorical feature with n possible values into n binary features, with only one active.Most of the ML algorithms either learn a single weight for each feature or it computes distance between the samples. Algorithms like linear models (such as logistic regression) belongs to the first category.



User	City
1	Roma
2	Madrid
1	Madrid
3	Istanbul
2	Istanbul
1	Istanbul
1	Roma

User	Istanbul	Madrid
1	0	0
2	0	1
1	0	1
3	1	0
2	1	0
1	1	0
1	0	0

One hot encoding example on City column

6. Grouping Operations

In most machine learning algorithms, every instance is represented by a row in the training dataset, where every column show a different feature of the instance. This kind of data called **“Tidy”**.

Tidy datasets are easy to manipulate, model and visualise, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table. — Hadley Wickham

Datasets such as transactions rarely fit the defi nition of tidy data above, because of the multiple rows of an instance. In such a case, we group the data by the instances and then every instance is represented by only one row.

The key point of group by operations is to decide the aggregation functions of the features. For numerical features, average and sum functions are usually convenient options, whereas for categorical features it more complicated.

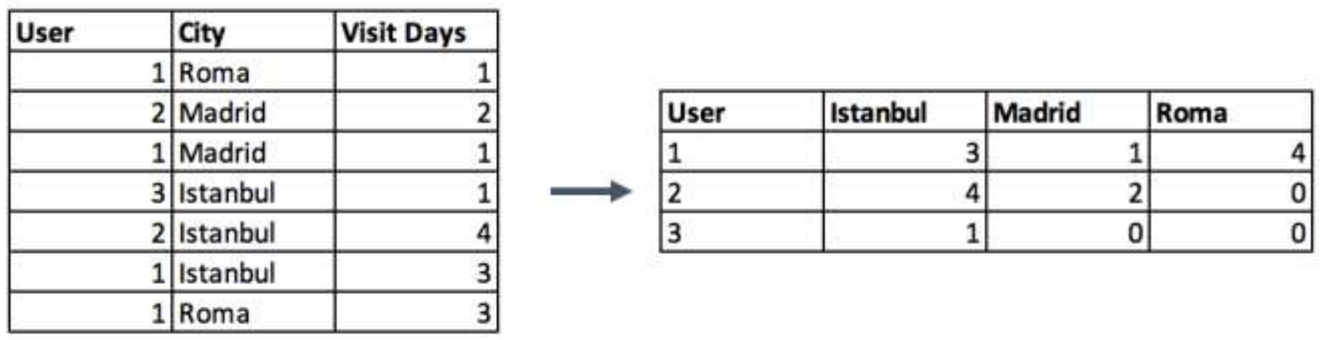
Categorical Column Grouping

I suggest three different ways for aggregating categorical columns:

- The first option is to select the label with the **highest frequency**. In other words, this is the **max** operation for categorical columns, but ordinary max functions generally do not return this value, you need to use a lambda function for this purpose.

```
data.groupby('id').agg(lambda x: x.value_counts().index[0])
```

- Second option is to make a **pivot table**. This approach resembles the encoding method in the preceding step with a difference. Instead of binary notation, it can be defined as aggregated functions for the values between grouped and encoded columns. This would be a good option if you aim to go beyond binary flag columns and merge multiple features into aggregated features, which are more informative.



User	City	Visit Days
1	Roma	1
2	Madrid	2
1	Madrid	1
3	Istanbul	1
2	Istanbul	4
1	Istanbul	3
1	Roma	3

User	Istanbul	Madrid	Roma
1	3	1	4
2	4	2	0
3	1	0	0

Pivot table example: Sum of Visit Days grouped by Users

#Pivot table Pandas Exampledata.pivot_table(index='column_to_group', columns='column_to_encode', values='aggregation_column', aggfunc=np.sum, fill_value = 0)

- Last categorical grouping option is to apply a **group by** function after applying **one-hot encoding**. This method preserves all the data -in the first option you lose some-, and in addition, you transform the encoded column from categorical to numerical in the meantime. You can check the next section for the explanation of **numerical column grouping**.

Numerical Column Grouping

Numerical columns are grouped using **sum** and **mean** functions in most of the cases. Both can be preferable according to the meaning of the feature. For example, if you want to obtain **ratio** columns, you can use the average of binary columns. In the same example, sum function can be used to obtain the total count either.

```
#sum_cols: List of columns to sum
#mean_cols: List of columns to averagegrouped = data.groupby('column_to_group')

sums = grouped[sum_cols].sum().add_suffix('_sum')
avgs = grouped[mean_cols].mean().add_suffix('_avg')

new_df = pd.concat([sums, avgs], axis=1)
```

7. Splitting Features

Splitting features using the Split Function. This is a good way to make features useful in terms of machine learning. Most of the time the dataset contains string columns that violates tidy data principles. By extracting the utilizable parts of a column into new features:

- We enable machine learning algorithms to comprehend them.
- Make possible to bin and group them.
- Improve model performance by uncovering potential information.

First, a simple split function for an ordinary name column:

```
data.name
0  Luther N. Gonzalez
1    Charles M. Young
2        Terry Lawson
3       Kristen White
4     Thomas Logsdon#Extracting first names
data.name.str.split(" ").map(lambda x: x[0])
0     Luther
1    Charles
2      Terry
3    Kristen
4     Thomas#Extracting last names
data.name.str.split(" ").map(lambda x: x[-1])
0    Gonzalez
1       Young
2      Lawson
3       White
4     Logsdon
```

The example above handles the names longer than two words by taking only the first and last elements and it makes the function robust for corner cases, which should be regarded when manipulating strings like that.

Another case for split function is to extract a string part between two chars. The following example shows an implementation of this case by using two split functions in a row.

```
#String extraction example
data.title.head()
0                    Toy Story (1995)
1                      Jumanji (1995)
2             Grumpier Old Men (1995)
3            Waiting to Exhale (1995)
4    Father of the Bride Part II (1995)data.title.str.split("(", n=1, expand=True)[1].str.split(")", n=1, expand=True)[0]
0    1995
1    1995
2    1995
3    1995
4    1995
```

8. Feature Scaling

In most cases, the numerical features of the dataset do not have a certain **range** and they differ from each other. Certain attributes tend to have higher magnitude than others. An example might be a person's income – as compared to his age. In real life, it is nonsense to expect **age** and **income** columns to have the same range.

Scaling solves this problem through ensuring the continuous features become identical in terms of the range. Comparable/ equivalent ranges.

This prevents your model from giving greater weightage to certain attributes as compared to others.

Basically, there are two common ways of scaling:

1. **Normalization**

Normalization (or **min-max normalization**) scale all values in a fixed range between **0** and **1**. This transformation does not change the distribution of the feature and due to the decreased standard deviations, the effects of the **outliers** increases. ***Therefore, before normalization, it is recommended to handle the outliers.***

2. **Feature Standardization**

Standardization (or **z-score normalization**) scales the values while taking into account standard deviation. This means standardizing data to have zero mean and unit variance. The features are rescaled so as to have properties of a standard normal distribution with $\mu=0$ and $\sigma=1$, where μ is the mean (average) and σ is the standard deviation from the mean.

If the standard deviation of features is different, their range also would differ from each other. This reduces the effect of the outliers in the features.

9. Decoding a Date-Time Column

A date-time column provides valuable information that can be difficult for a mode to take advantage of in it's native form such as the ISO 8601 (i.e. 2014-09-20T20:45:40Z).

If you suspect there are relationships between times and other attributes, you can decompose a date-time into constituent parts that may allow models to discover and exploit these ordinal relationships.

Here, I suggest three types of preprocessing for dates:

- Extracting the parts of the date into different columns: Year, month, day, etc.
- Extracting the time period between the current date and columns in terms of years, months, days, etc.
- Extracting some specific features from the date: Name of the weekday, Weekend or not, holiday or not, etc.

If you transform the date column into the extracted columns like above, the information of them become disclosed and machine learning algorithms can easily understand them.
And not to forget Time Zones. If your data sources come from different geographical sources, do remember to normalize by time-zones if needed.

Reframe Numerical Quantities

Your data is very likely to contain quantities, which can be reframed to better expose relevant structures. This may be a transform into a new unit or the decomposition of a rate into time and amount components.

You may have a quantity like a weight, distance or timing. A linear transform may be useful to regression and other scale dependent methods.

Creating new ratios and proportions

Instead of just keeping past inputs and outputs in your dataset, creating ratios out of them might add a lot of value. Some of the ratios, I have used in past are: Input / Output (past performance), productivity, efficiency and percentages. For example, in order to predict future performance of credit card sales of a branch, ratios like credit card sales / Sales person or Credit card Sales / Marketing spend would be more powerful than just using absolute number of cards sold in the branch

The Process of Feature Engineering

Feature Construction:

This involves the manual construction and handcrafting of new features from raw data done by the Practitioner. Feature importance and Feature selection informs you about the objective utility of features, but those features have to come from somewhere: Manually created. This requires spending a lot of time with actual sample data (not aggregates) and thinking about the underlying form of the problem, structures in the data and how best to expose them to predictive modeling algorithms.

This is the often talked about part of Feature Engineering that is attributed and signaled as the differentiator in competitive machine learning. It is manual, it is slow, it requires lots of human brain power, domain knowledge and it makes a big difference.

With tabular data, it often means a mixture of aggregating or combining features to create new features, and decomposing or splitting features to create new features.

With textual data, it often means devising document or context specific indicators relevant to the problem. With image data, it can often mean enormous amounts of time prescribing automatic filters to pick out relevant structures.

Feature Extraction:

This involves a host of algorithms that automatically generate a new set of features from your raw attributes. Some observations are far too voluminous in their raw state to be modeled by predictive modeling algorithms directly. Attempts to solve the problem of unmanageably high dimensional data into much smaller sets that can be modelled.

Common examples include image, audio, and textual data, but could just as easily include tabular data with millions of attributes.

For tabular data, this might include projection methods like Principal Component Analysis, Dimensionality Reduction and unsupervised clustering methods. For image data, this might include line or edge detection. Depending on the domain, image, video and audio observations lend themselves to many of the same types of DSP methods.

Feature / Variable/ Attribute Selection:

This is the automatic process of selecting a subset of relevant features (variables, predictors) that are most useful to the problem for model construction. Here, you are not creating/modifying your current features, but rather pruning them to reduce noise/redundancy.

Feature selection algorithms use a scoring method to rank and choose features, such as correlation or other feature importance methods.

More advanced methods may search subsets of features by trial and error, creating and evaluating models automatically in pursuit of the objectively most predictive sub-group of features.

Feature Selection Algorithms

There are three general classes of feature selection algorithms: filter methods, wrapper methods and embedded methods.

Filter Methods

Filter feature selection methods apply a statistical measure to assign a scoring to each feature. The features are ranked by the score and either selected to be kept or removed from the dataset. The methods are often univariate and consider the feature independently, or with regard to the dependent variable.

Example of some filter methods include the Chi squared test, information gain and correlation coefficient scores.

Wrapper Methods

Wrapper methods consider the selection of a set of features as a search problem, where different combinations are prepared, evaluated and compared to other combinations. A predictive model us used to evaluate a combination of features and assign a score based on model accuracy.

The search process may be methodical such as a best-first search, it may stochastic such as a random hill-climbing algorithm, or it may use heuristics, like forward and backward passes to add and remove features.

An example of a wrapper method is the recursive feature elimination algorithm.

Embedded Methods

Embedded methods learn which features best contribute to the accuracy of the model while the model is being created. The most common type of embedded feature selection methods are regularization methods.

Regularization methods are also called penalization methods that introduce additional constraints into the optimization of a predictive algorithm (such as a regression algorithm) that bias the model toward lower complexity (less coefficients).

Examples of regularization algorithms are the LASSO, Elastic Net and Ridge Regression.

Feature Selection in Python

Two different feature selection methods provided by the scikit-learn Python library are Recursive Feature Elimination and feature importance ranking.

Recursive Feature Elimination

The Recursive Feature Elimination (RFE) method is a feature selection approach. It works by recursively removing attributes and building a model on those attributes that remain. It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

Feature Importance:

This is an estimate of the usefulness of a feature that is helpful as a pre-cursor to selecting features. Features are allocated scores and ranked by their scores. Those features with the highest scores are selected for inclusion in the training dataset, whereas those remaining can be ignored.

The feature importance scores also provide information that can be used to extract or construct new features, similar but different to those that have been estimated to be useful.

A feature may be important if it is highly correlated with the dependent variable (the thing being predicted). Correlation coefficients and other univariate (each attribute is considered independently) methods are common methods.

More complex predictive modeling algorithms perform feature importance and selection internally while constructing their model. Some examples include MARS, Random Forest and Gradient Boosted Machines. These models can also report on the variable importance determined during the model preparation process.

Feature / Representation Learning:

This involves the automatic identification and use of features in raw data

Modern deep learning methods are achieving some success in this area, such as autoencoders and restricted Boltzmann machines. They have been shown to automatically and in a unsupervised or semi-supervised way, learn abstract representations of features (a compressed form), that in turn have supported state-of-the-art results in domains such as speech recognition, image classification, object recognition and other areas.

Pipelines, GridSearchCV and FeatureUnions in scikit-learn

The pipeline module of scikit-learn allows you to chain transformers and estimators together in such a way that you can use them as a single unit. This comes in very handy when you need to jump through a few hoops of data extraction, transformation, normalization, and finally train your model (or use it to generate predictions)

Pipelines help you prevent data leakage in your test harness by ensuring that data preparation like standardization is constrained to each fold of your cross validation procedure.

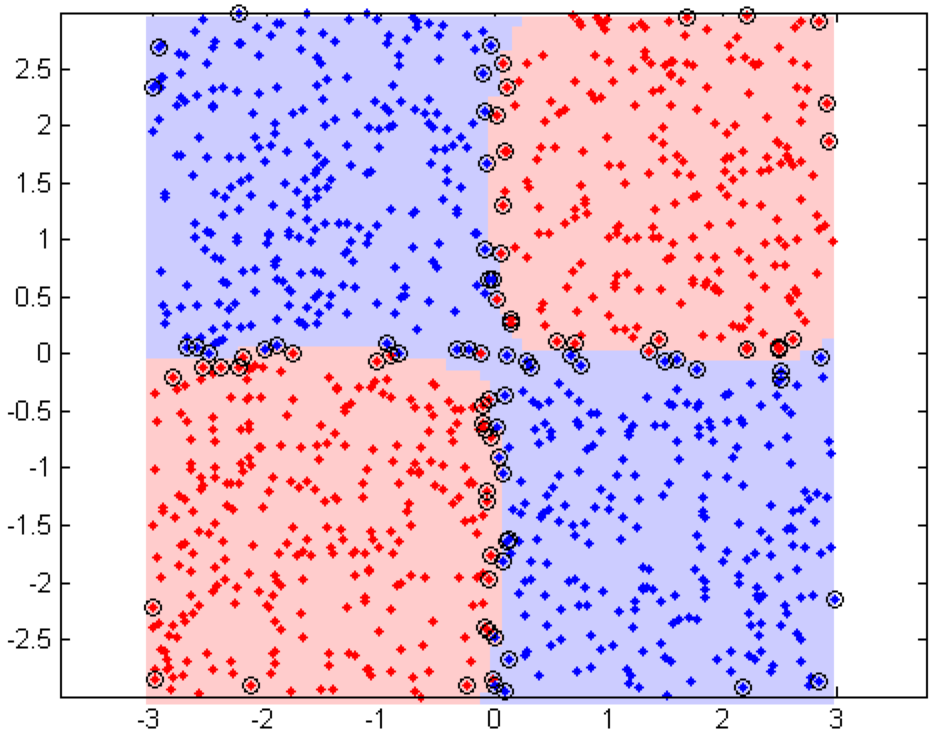
Iterative Feature Engineering for Machine Learning Pipeline

1. **Brainstorm features**: Really get into the problem, look at a lot of data, study feature engineering on other problems and see what you can steal.
2. **Devise features**: Depends on your problem, but you may use automatic feature extraction, manual feature construction and mixtures of the two.
3. **Select features**: Use different feature importance scorings and feature selection methods to prepare one or more “views” for your models to operate upon.
4. **Evaluate models**: Estimate model accuracy on unseen data using the chosen features.

Feature Crosses

Feature crosses are a unique way to combine two or more categorical attributes into a single one. This is extremely useful a technique, when certain features together denote a property better than individually by themselves. Mathematically speaking, you are doing a cross product between all possible values of the categorical features.

Consider a feature A, with two possible values {A1, A2}. Let B be a feature with possibilities {B1, B2}. Then, a feature-cross between A & B (lets call it AB) would take one of the following values: {(A1, B1), (A1, B2), (A2, B1), (A2, B2)}. You can basically give these 'combinations' any names you like. Just remember that every combination denotes a synergy between the information contained by the corresponding values of A and B.



All the blue points belong to one class, and the red ones belong to another.

Check out the following recommender materials/ articles and resources:

Feature Extraction and Image Processing for Computer Vision, Third Edition Book

There are a few videos on the topic of feature engineering. The best by far is titled **"Feature Engineering" by Ryan Baker.** Its short (9 minutes or so) and I recommend watching it for some good practical tips.

Automated Feature Engineering: Build Better Predictive Models Faster with Feature tools Library

https://www.analyticsvidhya.com/blog/2018/08/guide-automated-feature-engineering-featuretools-python/

Automating Machine Learning Workflows with Pipelines in Python and scikit-learn

https://machinelearningmastery.com/automate-machine-learning-workflows-pipelines-python-scikit-learn/

8 Proven Ways for improving the "Accuracy" of a Machine Learning Model

Get a sneak peak of these 8 proven strategies and ways to re-structure and enhancing model performance approach.

https://www.analyticsvidhya.com/blog/2015/12/improve-machine-learning-results/https://www.analyticsvidhya.com/blog/2015/12/improve-machine-learning-results/