**What are some best practices in Feature Engineering?**

I struggled with this question a lot in the recent times. The problem is that nobody explicitly tells you what feature engineering is. You are expected to understand for yourself what are good features.

I read up on Kaggle blogs and a variety of other forums and the theory and options are highly segregated.I feel this is because something which works very well for one Learning problem might bomb when we use it on other problems.

Take a look at the How to Win a Data Science Competition: Learn from Top Kagglers ([Advanced Machine Learning | Coursera](https://www.coursera.org/specializations/aml?siteID=lVarvwc5BD0-AqkGMb7JzoCMW0Np1uLfCA&utm_content=2&utm_medium=partners&utm_source=linkshare&utm_campaign=lVarvwc5BD0)) course in the Advanced machine learning specialization by Kazanova(Number 3 Kaggler at the time of writing)

So there is a lot to feature engineering, and I would try to accumulate all the ideas here. I would expect Quora users to correct me wherever I am wrong and help adding stuff as I would like to learn more about this.  
So here goes.

Things you can do with Features:

**1. Don't try predicting the future when you don't have to:**

If both training/test comes from the same timeline, we can get really crafty with features. Although this is a case with Kaggle only, we can use this to our advantage. For example: In the Taxi Trip duration challenge the test data is randomly sampled from the train data. In this case we can use the target variable averaged over different categorical variable as a feature. Like in this case Beluga actually used the averaged the target variable over different weekdays. He then mapped the same averaged value as a variable by mapping it to test data too.

**2. logloss clipping Technique:**

Something that I learned in the Neural Network course by Jeremy Howard. Its based on a very simple Idea. Logloss penalises a lot if we are very confident and wrong. So in case of Classification problems where we have to predict probabilities, it would be much better to clip our probabilities between 0.05-0.95 so that we are never very sure about our prediction.

**3. How best to use Latitude and Longitude features - Part 1:**

One of the best things that I liked about the Beluga Kernel is how he used the Lat/Lon Data. So in the example we had pickup Lat/Lon and Dropoff Lat/Lon. We created features like:

**A. Haversine Distance Between the Two Lat/Lons:**

1. def haversine\_array(lat1, lng1, lat2, lng2):
2. lat1, lng1, lat2, lng2 = map(np.radians, (lat1, lng1, lat2, lng2))
3. AVG\_EARTH\_RADIUS = 6371 # in km
4. lat = lat2 - lat1
5. lng = lng2 - lng1
6. d = np.sin(lat \* 0.5) \*\* 2 + np.cos(lat1) \* np.cos(lat2) \* np.sin(lng \* 0.5) \*\* 2
7. h = 2 \* AVG\_EARTH\_RADIUS \* np.arcsin(np.sqrt(d))
8. return h

**B. Manhattan Distance Between the two Lat/Lons:**

1. def dummy\_manhattan\_distance(lat1, lng1, lat2, lng2):
2. a = haversine\_array(lat1, lng1, lat1, lng2)
3. b = haversine\_array(lat1, lng1, lat2, lng1)
4. return a + b

**C. Bearing Between the two Lat/Lons:**

1. def bearing\_array(lat1, lng1, lat2, lng2):
2. AVG\_EARTH\_RADIUS = 6371 # in km
3. lng\_delta\_rad = np.radians(lng2 - lng1)
4. lat1, lng1, lat2, lng2 = map(np.radians, (lat1, lng1, lat2, lng2))
5. y = np.sin(lng\_delta\_rad) \* np.cos(lat2)
6. x = np.cos(lat1) \* np.sin(lat2) - np.sin(lat1) \* np.cos(lat2) \* np.cos(lng\_delta\_rad)
7. return np.degrees(np.arctan2(y, x))

**D. Center Latitude and Longitude between Pickup and Dropoff:**

1. train.loc[:, 'center\_latitude'] = (train['pickup\_latitude'].values + train['dropoff\_latitude'].values) / 2
2. train.loc[:, 'center\_longitude'] = (train['pickup\_longitude'].values + train['dropoff\_longitude'].values) / 2

**4. How best to use Latitude and Longitude features - Part 2:**

The Second way he used the Lat/Lon Feats was to create clusters for Pickup and Dropoff Lat/Lons. The way it worked was it created sort of Boroughs in the data by design.

1. from sklearn.cluster import MiniBatchKMeans
2. coords = np.vstack((train[['pickup\_latitude', 'pickup\_longitude']].values,
3. train[['dropoff\_latitude', 'dropoff\_longitude']].values,
4. test[['pickup\_latitude', 'pickup\_longitude']].values,
5. test[['dropoff\_latitude', 'dropoff\_longitude']].values))
6. sample\_ind = np.random.permutation(len(coords))[:500000]
7. kmeans = MiniBatchKMeans(n\_clusters=100, batch\_size=10000).fit(coords[sample\_ind])
8. train.loc[:, 'pickup\_cluster'] = kmeans.predict(train[['pickup\_latitude', 'pickup\_longitude']])
9. train.loc[:, 'dropoff\_cluster'] = kmeans.predict(train[['dropoff\_latitude', 'dropoff\_longitude']])
10. test.loc[:, 'pickup\_cluster'] = kmeans.predict(test[['pickup\_latitude', 'pickup\_longitude']])
11. test.loc[:, 'dropoff\_cluster'] = kmeans.predict(test[['dropoff\_latitude', 'dropoff\_longitude']])

He then used these Clusters to create features like counting no of trips going out and coming in on a particular day.

**5. How best to use Latitude and Longitude features - Part 3**

He used PCA to transform longitude and latitude coordinates. In this case it is not about dimension reduction since he transformed 2D-> 2D. The rotation could help for decision tree splits, and it did actually.

1. pca = PCA().fit(coords)
2. train['pickup\_pca0'] = pca.transform(train[['pickup\_latitude', 'pickup\_longitude']])[:, 0]
3. train['pickup\_pca1'] = pca.transform(train[['pickup\_latitude', 'pickup\_longitude']])[:, 1]
4. train['dropoff\_pca0'] = pca.transform(train[['dropoff\_latitude', 'dropoff\_longitude']])[:, 0]
5. train['dropoff\_pca1'] = pca.transform(train[['dropoff\_latitude', 'dropoff\_longitude']])[:, 1]
6. test['pickup\_pca0'] = pca.transform(test[['pickup\_latitude', 'pickup\_longitude']])[:, 0]
7. test['pickup\_pca1'] = pca.transform(test[['pickup\_latitude', 'pickup\_longitude']])[:, 1]
8. test['dropoff\_pca0'] = pca.transform(test[['dropoff\_latitude', 'dropoff\_longitude']])[:, 0]
9. test['dropoff\_pca1'] = pca.transform(test[['dropoff\_latitude', 'dropoff\_longitude']])[:, 1]

**6. Lets not forget the Normal Things you can do with your features:**

* Scaling by Max-Min
* Normalization using Standard Deviation
* Log based feature/Target: use log based features or log based target function.
* One Hot Encoding

**7. Creating Intuitive Additional Features:**

A) Date time Features: Time based Features like "Evening", "Noon", "Night", "Purchases\_last\_month", "Purchases\_last\_week" etc.

B) Thought Features: Suppose you have shopping cart data and you want to categorize TripType (See Walmart Recruiting: Trip Type Classification on [Kaggle](https://www.kaggle.com/c/walmart-recruiting-trip-type-classification/) for some background).

You could think of creating a feature like "Stylish" where you create this variable by adding together number of items that belong to category Men's Fashion, Women's Fashion, Teens Fashion.

You could create a feature like "Rare" which is created by tagging some items as rare, based on the data we have and then counting the number of those rare items in the shopping cart. Such features might work or might not work. From what I have observed they normally provide a lot of value.

I feel this is the way that Target's "Pregnant Teen model" was made. They would have had a variable in which they kept all the items that a pregnant teen could buy and put it into a classification algorithm.

**8 . The not so Normal Things which people do:**

These features are highly unintuitive and should not be created where the machine learning model needs to be interpretable.

A) Interaction Features: If you have features A and B create features A\*B, A+B, A/B, A-B. This explodes the feature space. If you have 10 features and you are creating two variable interactions you will be adding 10C2 \* 4 features = 180 features to your model. And most of us have a lot more than 10 features.

B) Bucket Feature Using Hashing: Suppose you have a lot of features. In the order of Thousands but you don't want to use all the thousand features because of the training times of algorithms involved. People bucket their features using some hashing algorithm to achieve this.Mostly done for text classification tasks. For example: If we have 6 features A,B,C,D,E,F. And the row of data is: A:1,B:1,C:1,D:0,E:1,F:0 I may decide to use a hashing function so that these 6 features correspond to 3 buckets and create the data using this feature hashing vector. After processing my data might look like: Bucket1:2,Bucket2:2,Bucket3:0 Which happened because A and B fell in bucket1, C and E fell in bucket2 and D and F fell in bucket 3. I summed up the observations here, but you could substitute addition with any math function you like. Now i would use Bucket1,Bucket2,Bucket3 as my variables for machine learning.

If you want to learn more about Data Science and Machine Learning, I would like to call out this [***excellent course***](https://www.coursera.org/learn/machine-learning?ranMID=40328&ranEAID=lVarvwc5BD0&ranSiteID=lVarvwc5BD0-btd7XBdF681VKxRe2H_Oyg&siteID=lVarvwc5BD0-btd7XBdF681VKxRe2H_Oyg&utm_content=2&utm_medium=partners&utm_source=linkshare&utm_campaign=lVarvwc5BD0) by Andrew Ng. This was the one that got me started. Do check it out.

Also, I will continue writing beginner friendly data science posts. Follow me up at [**Medium**](https://medium.com%2F@medium.com/@rahul_agarwal)

Some great pointers in the existing answers. I want to bring up one more way of thinking about feature engineering in addition to the a) business and b) data driven thinking: make it easy for your model to find the signal you suspect to be there. This is more about feature representation than engineering but can make a huge difference. The difference is in feature engineering you are looking to add more information into the feature space. In feature representation the question is more how exactly to represent the information. The below text is from an answer I gave to a related question:

If you have a universal function approximator (neural networks, decision trees, also read about VC dimensions if you want to know more) - a model that can express anything, and infinite data, you do not really need to worry that much about feature representation. This is basically what we see with the advances of deep learning. Decades of research on feature construction from images have become entirely obsolete….But in reality you usually do not have the luxury of ‘infinite data’.

And all of a sudden having a super powerful model class that can express anything is no longer the obvious best choice and instead you need to think about how to match the representation to the model class. In fact even for a universal function approximator, it is very beneficial to make it easier for them to find the answer (meaning the optimal gradient is nicely identifiable) and not get stuck in some local minimum on the optimization).

The answer on how to best represent features depends VERY MUCH on which type of model you want to use and the strategies for trees are for instance very different from those for linear models. Linear models are great at taking (numeric) differences, trees are terrible at it. Say you want a model to predict whether a company is profitable - this is simply a question whether revenue is greater than cost. If you have both features, this is really easy for a linear model and really hard for a tree to learn. So you can help the tree by adding differences of pairs of numeric features if you suspect they could matter. You suspect proportional relationships? Take logs and a linear model is done! Linear models on the other hand have a really hard time with nonlinear relationships (stating the obvious). Say in health you know that both being too heavy and too light is a problem - it is therefore a good idea to include the square of weight.

What about interaction effects (the infamous XOR problem)? Same story, you need to include pairwise products in the model to make it easy for a linear model to find, but a tree is good to go out of the box.

One example was the KDD CUP 2008 on a telecom churn prediction task with 50K anonymous features. Two members of our team independently tried to do a individual feature ranking and while we agreed somewhat, there were a number of features that had high mutual information and low AUC as individual contributors. This essentially means that we have a highly non-monotonic relationship. It turned out that in some features somebody had replaced missing values with some method and you could see spikes in the histogram. Those values with spikes were highly predictive but the linear model could obviously not learn from it - so we used decision trees on such single features as a means of discretizing the numeric feature and fed it into the linear models.

There are already some really good in depth answers to this questions, and I’m not sure I can add a ton but I’ll share a couple of thoughts.

In its essence feature engineering is the process of exposing the data in a form that a ML algorithm can take advantage of. That process can be broken out as follows:

Feature Engineering proper (as defined by myself): Use business experience and data driven insights to identify what in the data is correlated to the target.

Business driven features: This is where you should speak to the business and ask them what they think is predictive of the outcome. The business will give you, well, answers from a business perspective. Some might be great insights of things you would have not thought about, some might be very anecdotal and bring no value on an entire population, some you might not have the data to implement, some might require an approximation of the ideal data described by the business. Your job is to translate all of this into features.

Data driven features: this is where you try to add features that might not necessarily be something the business would associate to the target, but practically it ends up being a good predictor anyway. That might happen when a feature is really an approximation for something else harder to observe directly but correlated to the target. In classification problems I usually look at basic odds broken out by the class. If I see that my feature is present 20% of the time for one class but 90% of the time for the other class, I know I’m on to something.

The combination of those two should get you to a point where you can start running the data against a ML algorithm and see if indeed you have any predictive power going for you. If you are short of what would be satisfactory to the business you either go back an try to engineer more features or, if you already have exhausted the most promising ideas you can start digging into error analysis.

In error analysis I typically take observations that have been missclassified by the model and try to think about the why and look for insights that might lead to new ideas for features. Running these errors by the business is also a nice way to get a second round of input by providing something concrete for them to react to.

I usually start by looking at larger errors, and try to figure out how I can help my model prevent these situation. But it is not uncommon, especially in problems with low signal/noise ratio that your error anlysis might pay more dividends if done for smaller errors cases (i.e. can I bring a .4 prediction for class 1 up to .6?).

Once you are in the “range of acceptability” performance wise, you start a second process, that I think of as distinct from the feature engineering process I have just described, but is commonly also referred to as feature engineering (this is the most typical feature engineering done on Kaggle for example): feature optimization (again, as defined by myself).

Here you start thinking about minor tweaks to your features that might just expose the core signal in even better ways, thus allowing a specific model to take full advantage of it.

Remember that this process, in large part, is a function of the model you’re using. For instance, you’d do certain transformations on a feature to help a logistic regression that you wouldn’t do if you were using a random forest.

Examples of the type of things you’d do here are feature transformations (make the distribution of a feature behave more like a bell curve), dealing with outliers (do you trim them? do you create dummies to record which observations you trimmed?), sums, ratios, products of other features, binning of continuous features,…

To finish, make sure you spend most of the time on the first part of the process. There is really nothing in the second portion, or in the model you choose, or in hyperparameters tweaks that can substitute a great new feature.

This is an extremely important and extremely open-ended question.

The list below is by no means complete, but I hope it provides a good enough starting point.

Understand the data.

Feature engineering is an art. Understanding where do the features come from helps a lot.

Key questions:

Are the features continuous, discrete or none of the above?

What is the distribution of this feature?

Does the distribution largely depend on what subset of examples is being considered?

Time-based segmentation?

Type-based segmentation?

Does this feature contain holes (missing values)?

Are those holes possible to be filled, or would they stay forever?

If it possible to eliminate them in the future data?

Are there duplicate and/or intersecting examples?

Answering this question right is extremely important, since duplicate or connected data points might significantly affect the results of model validation if not properly excluded.

Where do the features come from?

Should we come up with the new features that prove to be useful, how hard would it be to incorporate those features in the final design?

Is the data real-time?

Are the requests real-time?

If yes, well-engineered simple features would likely rock.

If no, we likely are in the business of advanced models and algorithms.

Are there features that can be used as the "truth"?

If the problems to be solved would be of the supervised learning type, are there features that can be used as golden measures to be predicted?

Prices?

Yes/no decisions that have been recorded?

Recommendations followed or not followed?

Brainstorm and come up with as many diverse ideas as possible.

The best features are the ones that:

Can be intuitively explained.

Are always or almost always possible to compute, and

Either define some way of looking at the data extremely well or connect multiple ways of looking at the data extremely well.

Examples:

"Did the user register for the site?" is a good yes/no feature.

"How much time did the user spend on the site in the past week?" is a good continuous feature.

"How much time did the user spend on the site compared to the histogram of how much time did registered users spend on the site?" is a good feature combining both yes/no registration decision and continuous "time on the site" metric.

Always validate your guesses.

If there is the truth to predict, start building toy models from day one and establish a procedure to validate new models.

It would likely involve heavy Cross-validation (statistics) and resampling. Make sure you split the data into train and validation/test sets properly, so that potentially duplicate or similar examples are separated.

Make sure the model validation process is trustworthy.

Test it against some random features and/or random models to ensure that once your process detects a trustworthy model, you can trust in it.

Think back from the objective.

The end goal is rarely "look at this data and tell us what do you think of it".

Virtually always, one has some end goal in mind.

The goal might be estimating or predicting some parameter (Supervised learning, classification, regression), or it might be understanding the grouping and anomalies in the data (Unsupervised learning, clustering).

Often times, the end goal might give a hint on the cost functions that fit the problem best. This, in its turn, allows for better guesses on what types of models would perform best, even before the modeling has been started.

Keep RFMVT in mind.

RFMVT stands for:

Recency. Signals how old certain event is.

Frequency. Signals how often does certain events occur.

Monetary. Any numerical representation of direct of indirect business value of an example.

Variety. How many distinct items are found for certain type of an example.

Tenure. How much time has elapsed since the first appearance of certain example or of an example of certain type.

Don't overengineer.

The temptation is almost always high to build a big thing that can automatically assess how well certain feature, or type of features, can perform.

In the real life, during feature engineering phase, it does not make that much of a difference whether an iteration takes a few minutes or half a day.

The most productive time during the feature engineering phase is spent at the whiteboard. The most productive way to make sure it is done right is to ask the right questions about the data.

Seen in this light, feature engineering might well be the most academic and the most scientific part of what is called Big Data nowadays. There is no room for hacking; but there is no room for overengineering either.

Use the right tools for the right tasks.