How to do an ML project

Dominik Klepl

9/19/2019

Hey guys

So we've covered 8 weeks worth of lectures in just one week. **Crazy**. And now we have to apply all that stuff in our own project. Even more **crazy**.

From the people I've talked to I noticed that almost nobody has a clear idea how or even where to start with the project. Which is why I wrote this little guide/overview of the various stages of a proper ML project. For each stage I included a list of useful methods (some were covered in the lectures, some not)

In general, we all will have to go through the following:

- Find dataset - mostly **DONE**
- Project design
- Data preprocessing
- Explorative data analysis (EDA) + Visualisation (optional)
- Feature Engineering (optional)
- Feature Selection/Extraction
- Model training
- Hyperparameter tuning (optional & maybe advanced)
- Performance evaluation and model selection
- Interpret the results (optional/in some cases very advanced and complex)
- Write a paper
- Celebrate, sleep and celebrate again

1. Find dataset

Most of us have done this already but just for the sake of complete overview I included it here.

2. Project design

As aspiring data scientists we should learn to follow the common scientific practice. This step might be obvious for you but I feel like it's always a good reminder with the huge replication crisis that's currently happening in almost all scientific fields

Reference for those who have no idea what I'm talking about: Reproducibility crisis (https://towardsdatascience.com/the-reproducibility-crisis-and-why-its-bad-for-ai-c8179b0f5d38)

It's quite simple really, just write down every single step of the project after the preprocessing. Preprocessing is impossible to plan, something unexpected is going to happen for sure but everything else can be and should be outlined ahead of time. This way you won't be tempted to adjust your methods and therefore you won't risk producing a biased and potentially irreproducible paper.

3. Data preprocessing

When you first open your datasets you'll probably notice that it looks very different from the nice neat datasets like the iris dataset, it's messy, some samples have missing values, categories have various labels that in fact are the same category e.g. dog, puppy, doge, and so on. And no algorithm will be happy with such mess of a dataset.

If your dataset is clean than congratulations, you're either incredibly lucky or you managed to find a dataset that would never occur in real world.

Here's a list of various symptoms of messy data your dataset might be suffering of and some solutions for them:

3.1 missing values (=NAs) - Depending on how NAs you have, there are several ways to treat this (delete or impute) - If you have just a few NAs but otherwise large number of samples (rows) you can: - simply delete those rows - replace (=impute) the NAs with mean/median of that feature (column) or with mean of some meaningful subgroup - in case of time series use rolling mean/median - DON'T use on categorical features - it's fast and easy but not very accurate - If there's more NAs and/or you have a limited number of observations (rows): - replace the NA with the most similar value - this can be done 'manually' or run a simple KNN algorithm (K=1) - train a model (linear regression, random forest, SVM,…) using all the other features (don't use the target variable) to predict the feature with NAs - use Multivariate Imputation by Chained Equation (MICE) - for details see: Research paper (https://www.jstatsoft.org/article/view/v045i03/v45i03.pdf) - Python implementation of MICE in package impyute (https://github.com/eltonlaw/impyute) - Deep learning imputation - very advanced stuff but effective on both numerical and categorical features - Python implementation in package Datawig (https://github.com/awslabs/datawig)

3.2 feature is in different dataformat than it should be This one's easy. If your program (Matlab, Java, Python, R or whatever you use) says a feature vector is a string (characters) when it should be numerical just change the class of that object.

3.3 Features have weird names If your brain is simply unable to remember that that feature Answer42 is the amount of stress just rename it.

I could go on and on with this list. Simply clean your data so that both you and your algorithms are happy to work with such data :-)

Finally, divide your data into training and testing sets, the ratio is up to you. N.B. when doing model-based NA imputation train the mode only on the training set, otherwise you'll 'stain' your training data as they wouldn't be entirely new to the model.

4. Explorative data analysis and Visualisation

In this phase just get to know your dataset. **Go nuts.** It's good to begin with looking at basic descriptive statistics of your features and target. Look at the mean, standard deviation etc. to familiarize yourself with the data.

Also look at how balanced your target variable is (if you're doing classification). If there's a lot more samples of class A than class B than your model might learn the pattern that class A has higher probability to occur in new data and it will simply predict all new values as A.

There are several solutions, I'll include 4 here: umsampling and downsampling. Upsampling means that you randomly duplicate some of the samples of the underrepresented class until you reach balance. Upsampling (random over-sampling) has the advantage that you're not throwing any data away, on the other hand you're assigning more weight to the upsampled data (the model will assume those values are more likely to occur than others). An alternative is using SMOTE or ROSE which is informed over- and undersampling Downsampling (random under-sampling) is the exact opposite, randomly delete samples of class A until you reach balance. This method removes the disadvantage of upsampling but you're loosing data. If your dataset is small than it's not really preferable. Of course there is always an option to use algorithms that aren't balance-sensitive i.e. they assign different cost to the minor class. Another option are the ensemble based methods.

Next, you can plot each of the features with your target variable (=y) to see if you can eyeball some interesting pattern. Or plot 2-3 features, you might fight an some relationship between that which you can later use to create some sexy new features.

Furthermore it's usually a good idea to check the distribution of all the features. Some models have built-in assumptions about the shape of distribution such as linear regression which eats only normally (Gaussian) distributed features and target. If the features don't have the distribution you'd imagine them to have then transform them.

——– BONUS - how to reach normality ——– If the distribution has the bell shape but it's skewed i.e. pushed to one side then calculate the square root of the observations. If you have an exponential distribution i.e. log-normal distribution a natural logarithm of the observations will do the trick. In case you don't feel confident enough to decide what kind of transformation would fix your distribution use Box-Cox method which essentially decide the best fit for you, it has a lambda parameter that indicates what transform will be applied. Simply iterate over all of its values, plot the resulting distribution and select the most bell-shaped.

Below are some common values for lambda: - lambda = -1. is a reciprocal transform. - lambda = -0.5 is a reciprocal square root transform. - lambda = 0.0 is a log transform. - lambda = 0.5 is a square root transform. - lambda = 1.0 is no transform.

This is by no means an exhaustive list of ways to achieve a normal distribution. Further problems can be long tails, outliers and many more.

Instead of plotting all the features you can also run a correlation analysis which can again indicate some interesting relationships between the features or it might tell you that some dimensionality reduction (e.g. PCA) might be a good idea.

Finally, it is good practice to normalize your features. Sometimes the model can handle the original data but if it can't it can be difficult to identify that issue.

5. Feature engineering

This step is an optional step for several reasons: You don't want to. There are no features to create. The problem requires domain knowledge to be able to create new features.

Feature engineering essentially means to craft new features. Either you can combine 2 or more features into a new, e.g. given features of travelled distance and time you could create a feature for speed. Or a single feature contains more information that can be extracted into a new one e.g. feature date can be split into year, month and day. Feature engineering can often dramatically improve the performance of your models so try to think about it.

6. Feature selection/extraction

There are essentially two categories of feature selection: filtering and model-based (forward or backward).

Filter methods are for example zero and near zero variance filters which remove features that have very low variance or low proportion of unique values.

Model-based - see lecture 5.

Next you might consider applying some feature extraction method if you have very large number of features and/or the EDA revealed high intercorrelation.

7. Model training

Alright, you've finally got to the sweet realm of model training. This one is quite straightforward really. Just pick a few algorithms that you think are well suited for your problem and feed the training data to them.

One thing to consider before doing so is some form of cross-validation (CV). CV improves the accuracy of performance measurement since it's basically a simulation of several rounds of training and consequent testing on previously unseen data. You also have to decide what kind of CV to use, leave-one-out, k-fold, repeated k-fold. Another thing to consider is preventing 'data leakege'. For example if you have a dataset about football players and their performance across multiple games. In other words, you have multiple datapoints coming from the same player. In such case you need to ensure that data from the same subject don't 'leak' across the train-test data split or in case of k-fold CV across the folds. In such, you can forget about the leave-one-out CV, it's useless. Why is preventing leaks important? In short, the resulting model would be biased because your testing data isn't really never-seen-before data since the model already saw data about that subject.

Now, what are some of the algorithms available to throw at your problem? (bold = covered in lecture)

7.1 Classification

- **Decision tree**
    - learns smart data splits to classify the output
- Random Forest
    - extension of decision tree, let multiple trees grow - more powerful and flexible than decision tree but also more computationally expensive
- **KNN**
- looks for k most similar datapoints
- lazy learner - fast training (no training, just save data) but resulting model is large (memory-wise)
- **Logistic regression** - Gaussian assumption
    - fit linear regression which is in log-odd space, use sigmoid to transform the line into probabilities
    - to make sense of the parameters take logit(a) to obtain probability - each parameter says by how much the probability of positive class increases given the feature value
    - in the output of the model, it's important to look at significance (i.e. p-value) of each of the effects (=feature parameter/theta) if p-value>=0.05 than it suggests that that effect won't be present in new data - it's a bit more complex than this but I'm sure we'll cover that in the Intro to statistics course ;-)
- Bayesian logistic regression - based on Bayes theorem and bayesian

approach to statistics, counterintuitive to understand but super hype these days (at least in statistics) - Gaussian assumption
    - in short, it doesn't rely on p-values
- **Linear Discriminant Analysis**
- **Support Vector Machine**
- finds a line that divides the classes the best i.e. maximizes the distance from both classes
- **Naive Bayes** - Independence assumption, very sensitive to class imbalance
- simply count occurences of values in the sample, class with the highest resulting posterior probability is assigned to the datapoint
- (Deep) Neural Network
- I'm so looking forward to this, just 3 weeks :-D
- Perceptron
- XGBoost (E**x**treme **G**radient **Boost**ing) - Of course 'no free lunch' theorem is true but if there were to be a counterargument, XGBoost would be it!!!
- one of the variants of boosting ensembles
- the Kaggle competion winners usually use them (of course except for deep learning models)

7.2 Regression

- **Decision tree**
- Random Forest
- **Linear regression** - Gaussian assumption
- Bayesian linear regression - Gaussian assumption
- **regularized** regression: Lasso, Ridge, Elastic net
- punish the model for using too many features, therefore parameter value will be proportional to the information gain of given feature
- **Support Vector Machine** - it's called differently, can't remember now
- (Deep) Neural Network
- Perceptron
- XGBoost

7.3. Clustering

- **k-Means**
- **k-Medoids**
- Fuzzy C-means
- Self-Organizing Maps
- **Hieararchical Agglomerative Clustering**
- Gausian Mixture model

8. Hyperparameter tuning

I don’t assume that many of you will use this step now but again to satisfy my perfectionism I’m including it here. Hyperparameter tuning simply means to train multiple versions of the same model but you vary the parameters of the model in each iteration. In other words, you’re trying to find the best parameter values and/or best combinations of parameters. For example if you train a regularized regression try various values of the regularization parameter. Then evaluate the best fit using some predetermined metric such as sensitivity, f1 score, rmse, information criterion (set of evaluation metrics based on information theory).

9. Performance evaluation and model selection