Data Science Competitions

Week 4

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

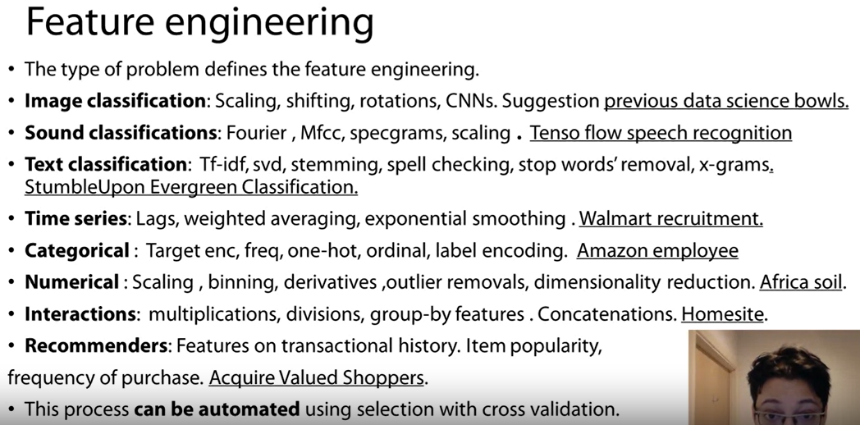
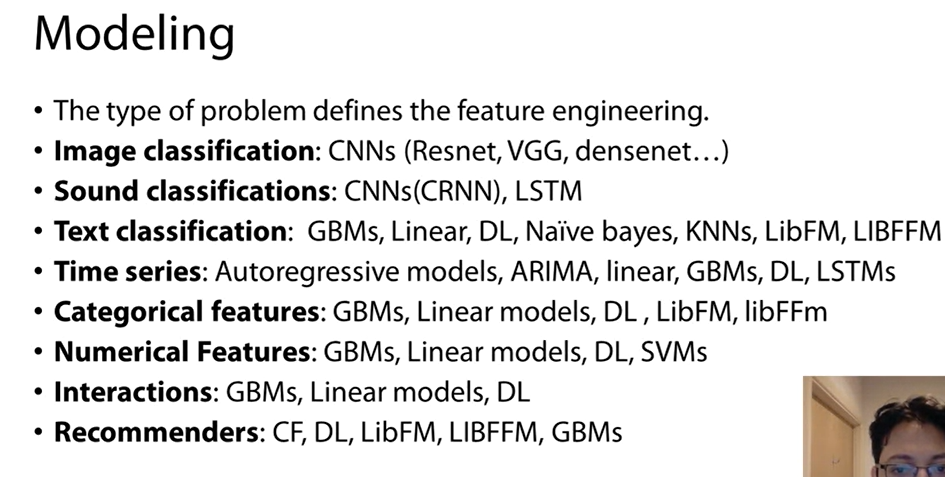
Hyperparameter Tuning

* Libraries
  + Hyperopt
  + Scikit optimize
  + Spearmint
* Tree-based Models
  + LightGBM, XGBoost, CatBoost, sci-kit learn, RGF
    - RandomForest builds trees independent of each other, while Boost models don’t > Many trees do not lead to overfitting for RF
  + Max-depth – If you increase & validation score keep improving, it shows that there is a lot to be extracted from the data – but takes more time
    - N\_estimators for RF – start with a small value and gradually increase (no limit here as RF does not overfit)
    - ExtraTrees: Can be set to none == no limit for how many tres
  + LightGBM: Num\_leaves – high depth but low num\_leaves can avoid overfitting
  + Subsample / baggin\_fraction: Only use a fraction of objects at each iteration
  + Colsample by tree & colsample by level: Only use fraction of features to avoid overfitting
  + Additional Regularization: Min\_child\_weight, lambda, alpha // min\_data\_in\_leaf, lambdal1, lambdal2
    - Min\_child\_weight one of the most importantparameters to tune – might have a wide range of values from 0 to 300
  + Eta – Learning rate
  + Num\_round – How many learning steps we want to perform == How many trees we want to build
    - Set eta to be very small & then find how many rounds until we overfit w/ early stopping (exit at minimum of val loss
      * > Then scale eta & iterations by a factor alpha >>> E.g. divide eta by half & double the iterations
  + & specify a seed!
    - Good to checkout how dependent a competition is on randomness > E.g. see the variance from one to another random seed!
    - If this variance is very high – perhaps not a good competition to participate in as results will be very random // adjust validation to account for randomness
  + Criterion: gini vs entropy – try both
  + N\_jobs: how many cores you have
* NNs
  + Dropout shouldn’t be in the very first layer, as then some information is lost completely – the later the more
  + Batch size & learning rate should both be scaled by the same factor alpha
  + Start with big learning rate > Decrease from experiment to experiment
  + Static Dropconnect
    - Dropconect: Dropping random connections
    - Static: Dropping one pair of random connections and keeping it the same throughout all learning iterations
    - > Hidden size of layer to whom connections are dropped should be big
* SVMs & Linear Models
  + LogisticRegression, LinearRegression, SGDClassifier, SGDRegressor
  + Do not require extensive hyperparameter tuning
  + Via liblinear & libSVM
  + Vowpal Wabbit – FTRL for datasets which do not fit into memory, as it only loads row by row into memory
    - Also available implementation in pure Python
  + Regularization parameters (C, alpha, lambda)
    - Start with very small C e.g. 10^-6 > Increase by factors of 10
    - Try both L1 & L2
      * L1 has the advantage of weight sparsity, hence good for feature selection
* You dont win tuning hyperparameters!
* Be patient with model training!
* Average everything!
  + Predictions from multiple random seeds
  + From different hyperparams e.g. average results from max depth = 4, 5 , 6 to get a better 5 than just 5

Tips & Tricks

* Define goals – Money, Skills
  + E.g. if interested in applying ML to Medicine, participate in one of the medical challenges
* How many submissions are people taking?
  + If people with few submissions in top ranks, there should be a trivial approach to the competition // leaks discovered only by few people
  + If many submissions in top – perhaps leaderboard probing / bad correlation of train & test
* Understand why or why not your ideas work
* Rate all possible parameters by
  + Importance – May have a big impact
  + Feasibility – May take long to get good results
  + Understanding – I know that the param is doing
* Data Loading – Basic preprocessing to save time lateron
  + Convert csv/text to hdf5 (for pandas)/ npy(for numpy) for faster loading
  + By default data is stored in 64-bit arryas (8 bytes) – most of the time can be downcasted safely to 32 bits
  + Large datasets can be processed in chunks
* Extensive CV only when it is needed at the very end for marginal improvements
* Start with the fastest models & early stopping
* Use VCS – Version Control System, e.g. via Git or Kaggle
* Get domain knowledge – read scientific articles on the topic
* Keep Code clean & reproducible
* One notebook per submission
* Store dev as separate csv & then just switch paths for full training
* Macros
* Own library with frequently used functions & training code – saves a lot of time

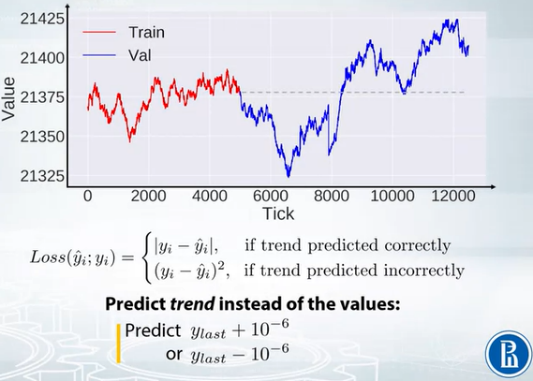
Competition Pipeline:

* Add. Pipeline: <https://github.com/Far0n/kaggletils>
* Understand the problem – 1 day
  + Type of problem
  + How BIG is the data?
  + Hardware needed (CPUs, GPUs, RAM, Disk-space)
  + Software needed (Torch, sklearn, GBM)
  + Setup Anaconda environment / Virtualenv
  + What is the metric?
  + Previous Code to copy paste?
* EDA – 1-2 days
  + Histograms, Correlations
  + Plot features vs target & vs time
  + Crosstabs
  + Differences in how a feature is in the train vs test set?
  + Bin numerical features > Are there any non-linearities?
* Define CV Strategy
  + People have won just by selecting the right CV Strategy
  + Is time important? > Split by time
    - Predict future data with train, ideally use the same time-split as testing data
  + Different entities in train & test data? > Stratified validation
    - E.g. different customers in train vs test data
    - Stratified validation e.g. given to us for free in HM
  + Is it completely random? > Random validation e.g. K-Fold
  + Sometimes combination of the above
  + Prove-test your validation with the leaderboard
    - Does your val score correlate with leaderboard score?
    - Does a different val give you a better score?
* Feature Engineering – Until last 3-4 days
  + Different problems require different feature engineering
  + You can automate it quickly if you have a good CV strategy, as you can then validate them quickly
* Modelling – Until last 3-4 days
* Ensembling – Until last 3-4 days
  + Save predictions for both validation & test as .csv
  + Then try out what combination of different model’s csv’s work best – e.g. an average, weighted average, multilayer stacking
  + Smaller data tends to require simpler ensembling techniques like averaging
  + Look at the correlation of the two models predictions – the lower the correlation the better the performance if you combine them (Pearson correlation)
  + The stacking process repeates the modelling process, as it is like having a new set of features >> You can re-apply the same process as in modelling & experiment with combinations
  + Select submissions with no too strong correlation & perhaps one best on leaderboard & one best on private validation
* Try it yourself shut from open kernels for one week to get an intuitive approach

Week 3

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

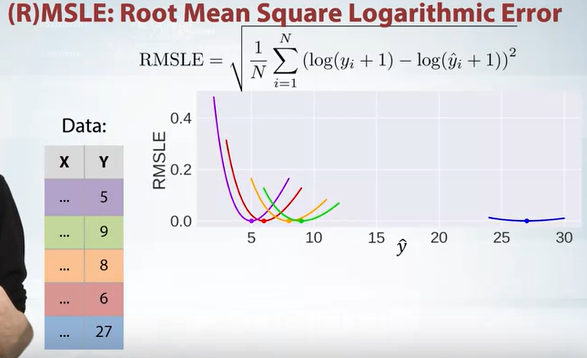
*Regression Metrics*



Time-series metric example:

* Instead of predicting the next value from ground-up, instead built a model to predict whether it’s the previous value + tiny constant or previous value – tiny constant

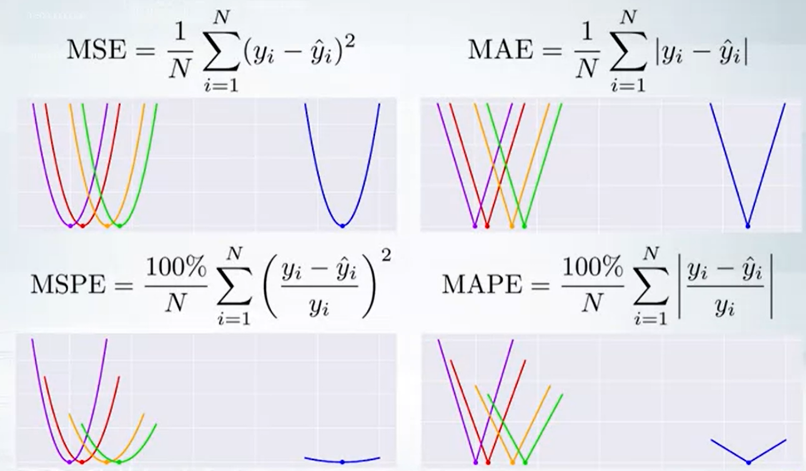
(R)MSE / MSPE (Mean squared percentage error)

* Root of MSE
* Scaling it back after the square
* Difference only when using gradient-based methods, but generally a model better on MSE is also better on RMSE
* Best constant for RMSE/MSE: Mean of target (Weighted for MSPE)
* RMSLE – Root Mean Squared Logarithmic Error
  + Same relative curves as MSPE, but asymmetric, larger errors are perceived to be better
  + Solves the small bias problem of MSPE & MAPE

R^2

* Scores MSE between 0 & 1 (the higher the better)

MAE / MAPE (Mean absolute percentage error)

* Less sensible to outliers
* Usually used in Finance, as 10$ error is just 2 times worse than 5$ error, not 4 times
* Best constant: Median of target (as outliers matter less) (Weighted for MAPE)
* First derivative is the step function
* Use if you want to trait outliers equally
  + If we want to penalize outliers -> MSE

*Classification Metrics*

Accuracy

* Best constant value = most frequent class
  + Makes it easy to bias if dataset is biased
* Difficult to optimize, as only cares about hard values (argmax), not actual predictions

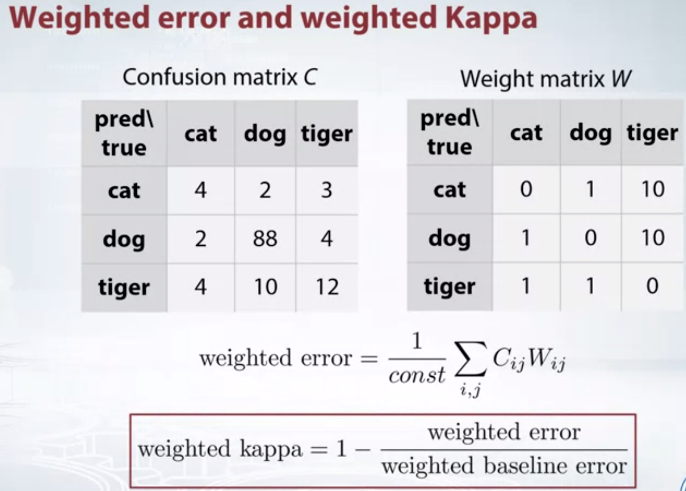
Log Loss

* Best constant value = set alpha(i) to frequency of class(i)
* Uses soft values
* Strongly penalizesvery wrong answers > Prefers a lot of small mistakes than one very serious mistake

AUC

* Uses soft values to rank predictions
* Can be calculated as # Correctly ordered pairs (e.g. starting from left ) / # total pairs
* Best constant value: - Does not matter; All constants lead to the exact same base line
  + 0.5
  + That’s makes AUC great & No 1 choice for your own experiments

Cohen’s Kappa

* Refactor accuracy
  +  e.g. in a dataset where base accuracy is 0.9 -> 0 & 1 -> 1
* Baseline is set at the average accuracy achieved with random predictions multiple times

Weighted Error:

* Multi-classification & the error for predicting one class instead of another is set to be higher
* Calculated by multiplying a confusion matrix of our predictions times a respective weight matrix
* Quadratic Weighted Kappa

Loss and Metric:

* Target Metric – WE want to optimize
* Loss Metric – MODEL optimizes
* Optimizing the Metric:
  + Target Metric == Loss Metric
    - MSE
    - LogLoss
  + Pre-process train set & optimize other metric
    - MSPE, MAPE, RMSLE
  + Optimize other metric & post-process
    - Accuracy
    - Kappa
  + Write custom loss function
    - Quadratic weighted Kappa
  + Optimize another Metric, monitor desired Metric, early stop

Regression metrics:

* MSE > Possible to optimize with almost every library; NNs
* MAE > XGBoost does not work, as no Derivcate; LightGBM & other libs work; NNs
  + Sometimes called Quantile loss, L1 metric, L1 loss; Huber Loss (Mix between MSE & MAE depending on how big the value is)
* MSPE & MASE
  + > As weighted versions of MSE & MAE, we can use MSE & MAE and feed an additional weight into it (many libs accept sample weights)
  + > Sample from the DF with weights yourself!
  + > Logarithmic optimization similar to Kaggle rossman challenge
* RMSLE – easy as similar to MSE
  + > Transform target variables to z scoresz = log(y + 1)
    - > Then fit target to MSE loss
      * > At test transform back to original via exp(z) – 1

Classification metrics:

* Logloss > Many libs, NNs by default normally, RFs a bit more tricky
  + > Predictions should be calibrated to closely follow the target rolling mean
    - > Platt scaling (Fitting a logistic regression onto our outputs)
    - > Isotonic regression (Fitting a isotonic regression to output)
    - > Stacking (Just use any model & e.g. evaluate it with ROC AUC, then take the models predictions before submission and calibrate them properly)
* Accuracy > Fit any metric & and then tune with the binary threshold; Hyp tuning
  + Difficult as not differentiable (Zero-one loss)
  + > Threshold tuning is key
* AUC ROC > Use specific algorithm to optimize
  + Despite non-differentiable like Accuracy, we can optimize AUC-ROC with a specific algorithm
  + > Pairwise Loss should be defined i.e. if a pair is ordered correctly good, else no
    - > Avlble in XGBoost, Light GBM, NNs via github
      * Many people still use LogLoss instead of pairwise loss as similar results
* Quadratic Kappa
  + > Optimize MSE & then find right thresholds
  + > Implement custom smooth loss for NNs or GBDTs (See Notebook)

Mean Encodings

* Generating features based on the target
  + Simplest: mean of all target values
    - Mean encoding instead of label encoding can help trees become shorter & better – the more complicated & complex the label to encode, the better (e.g. cities)
* Validation is very important
* Types:
  + Likelihood: Mean of target(as above)
  + Weight of evidence: Ln of mean of target
  + Count of goods(sum)
  + Diff between number of ones & zeros
* Require regularization to avoid overfitting on train!
  + CV loop inside training data
    - Only base the encoding on the rest of the subset
    - 4-5 folds for good results; no further tuning needed
    - Need enough data
  + Smoothing
    - Punishing low-frequencies
  + Random noise
    - Requires lots of finetuning & time
    - Unstable
    - Often used together with LeaveOneOut Reg
  + Sorting & Calculating expanding mean
    - Only use some rows of the data to calculate the mean for N
    - CatBoost library
* More possibilities for regression tasks as we can use median, mode & other ‘mean encodings’
* Many-to-many relations: Turn the many-to-many relations into multiple rows with partly duplicates
* If you participate in any competition with many categorical labels, you should always try mean encodings
  + Never estimate encodings before tr & val split

Week 2

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

**Feature Preprocessing & Generation:**

**Numerical Features:**

*Feature Scaling:*

Why:

* Most ML Algorithms, such as kNN use Euclidean Distance btw points (-> = Ordinary straight line distance btw two points in a Euclidean Space (any space rly))
* Example: If Age is e.g. on a scale of 0 - 80 & SibSp is on a scale of 0-6, & we have point 40, 0, who we classify as died and a point 80, 0 who died as well, but our algorithm might classify it as not died since there is such a huge distance of 40 in age! -> Another point at 40, 2 however might be classified as died since the diff in sibsp is only 2 (which is huge for sibsp, but nothing for age)

How:

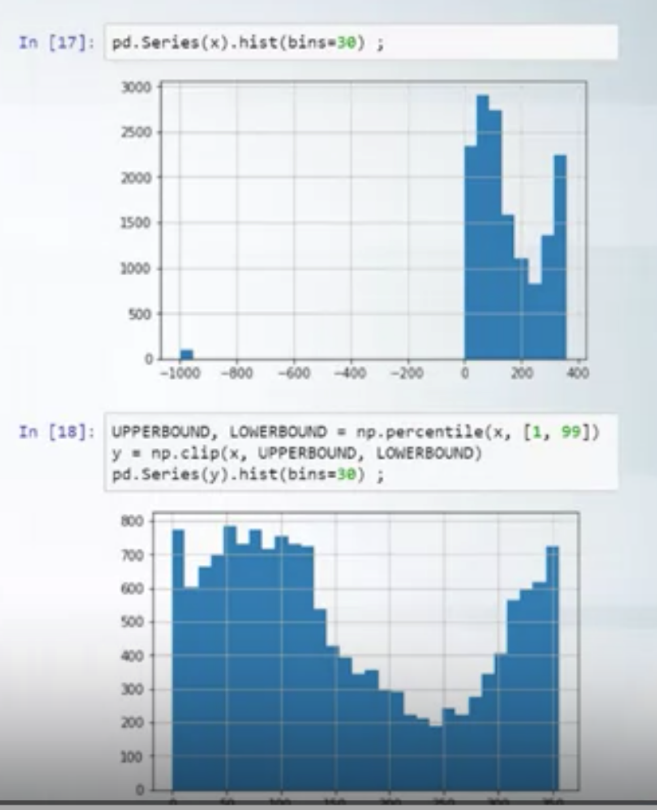
* Min-Max Scaling (esp. for KNN, PCA, Grad Descent:
  + sklearn.preprocessing.MinMaxScaler
  + Scale to [0, 1]
* Unit Vector (esp. for Regularization)
  + Divide values by their unit vector
  + Scale to [0, 1]
* Standard-Scaling (esp. for Lin/Log Regression):
  + sklearn.preprocessing.StandardScaler
  + Scale to mean=0 & std=1
* Note these cannot be used to deal with outliers, as the relative distances will remain! -> Deal with outliers first

When:

* k-nearest neighbors
* Principal Component Analysis
* Gradient Descent (Esp. to speed it up!)
* Not necessary for:
  + Tree-based models
  + Naive Bayes
  + Linear Discriminant Analysis

*Outliers:*

* Skew the model unnecessary in one direction despite only being e.g. one of thousand values
* Clipping / Winsorizing
  + Clip data between lower & upper bound -> The bounds could be percentiles of the feature, such as the 1 & 99% percentile
  + Esp. used for financial data



* If our outliers matter:
* Rank:
  + sorting a feature by their values and then transforming them to their index
  + (Make sure to concat train & test data when applying the rank transformation!)
  + scipy.stats.rankdata
* Log Transform /  Square Root Transform - esp. for NNs:
  + e.g. np.log(1 + x) -> Values near 0 become more distinguishable
  + Drives big values closer together
  + Or sqrt transform if all values are positive

*Feature Generation:*

* Examples:
  + House Price + Total sqm -> Add price/sqm
  + -> Calculate cubic size of object (Trees often have problems with multiplications; Try to take them away)
  + Fractional parts: E.g. for prices if they are 2.99 and 6.50 add 0.99 and 0.50 as features -> Uses differences in price perception /// To distinguish machine & human -> Humans would never take exactly 2s to read a facebook message, but some fraction at the end!

**Categorial Features**

*Numeric vs Categorical features:*

* Pclass would be a numeric feature if the difference between Pclass 1 & Pclass 2 would be the same

*Ordinal vs Non-Ordinal*

* e.g. S, Q, C vs low, middle, high

*Label Encoding:*

* Map categorical features to different numbers
  + Works well for tree-based model
  + Encode by order of appearance if meaningful
  + Alphabetical order
  + Order of frequency
  + sklearn.LabelEncoder
  + pandas.factorize
* Standard labelencoding for kNN & NN works badly
* Frequency encoding with all values summing up to 1, could work well for kNN & NN if the frequency is meaningful
  + Some diff type of label encoding should be applied if the frequencies of two are the same or very similar!
* One-hot encoding
  + Ordinality might be lost
  + If only two values, OHC & LabelEncoding will be ~same
  + Good if the labels are very independent from each other (i. e. a non-linear non-ordinal relationship)
  + Works well for NN, Linear Models & kNN
  + Tree-methods will slow down and results may not improve)
  + In case it results in many columns make it a sparse matrix by storing only non-zero values - -
    - (<https://stackoverflow.com/questions/40817459/xgboost-and-sparse-matrix>)
    - Make sure the proportion of new columns by one-hot encoding is not too big (-> If you have 9 important numeric features and 1 categorical feature with 100 diff. values then you'll end up with 109 columns and the 9 numeric features will lose attention in the models)
  + pd.get\_dummies
  + sklearn.preprocessing.OneHotEncoder

*Feature Interaction*

* esp. useful for non-tree models such as kNN & linear models
* Combining e.g. sex & pclass
  + 1male, 2male, 3male & 1female, 2female, 3female
    - And then one-hot-encode
  + Linear model would adjust its predictions for each of these possibilities and get a better result

**Datetime & Coordinates**

*Datetime:*

* Periodicity
  + Capture repetition in data
  + e.g. once a day, 3x a minute
* Time since:
  + Row independent:  One starting value counting for all values (e.g. from year 2000)
  + Row dependent: -> Each row's scale is different (e.g. days left until birthday; Days passed since last weekend)
* Difference between dates:
  + e.g. in churn prediction -> Date customer has subscribed; date customer cancelled;
  + After calculating the difference and getting a day difference you might want preprocess this new generated feature again or encode it

*Coordinates:*

* Dealing with coordinates can be very tricky,  as there are many possible features to generate
* Distances: e.g. for flat prices, distance to old buildings; distance to most expensive flat in grids of 250sqm etc; Distance to central clusters
* Aggregate Statistics: Mean amount of old buildings in area; etc
* For decision trees, it might help to add features with slightly rotated coordinates

*Handling Missing Values*

* Can be in the shape of:
  + NaN (Not a number)
  + Empty String
  + -1 ; -9999; 99; 999
* Find out by plotting a histogram of the data and looking for weird outliers

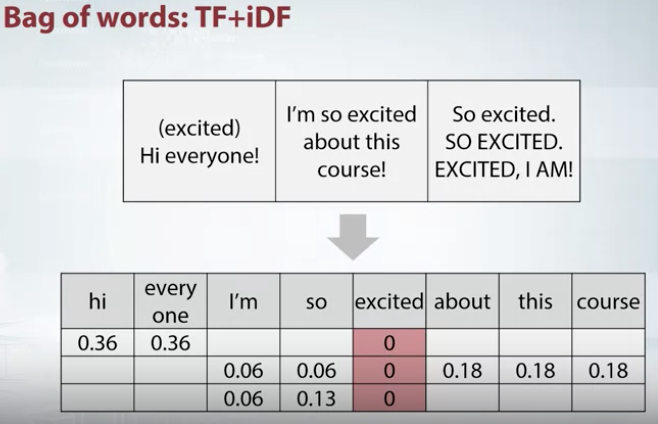
*Imputation:*

* Replace with value outside feature range
  + -> Tree-based models could put this in a sep. category; Linear Models & NNs might suffer
* Replace with mean/median
  + ->  Better for Linear Models & NNs; More difficult for Tree-based models
  + Be careful with e.g. temperatures (accidentally putting in the median temp. in winter)
* Reconstruct value
  + -> Generate a new feature, saying whether or not the value is existent (i.e. an isnull feature)
    - Can help both NNs & Trees, but doubles no of columns
  + Reconstruct e.g. in a time-series from nearby values
  + Train a model to predict the value you want to reconstruct
* Be careful with feature generation after having imputated values (i.e. mean or other values might be biased)
* XGBoost and other models can sometimes handle NaN!
* You can also turn outliers into missing values if the outliers make no sense & the missing value is more meaningful

**Feature Extraction from Text & Images**

**Text:**

* Bag of Words
  + Column for each word in vocab and respective counts as values
  + sklearn.feature\_extraction.text.CountVectorizer
* Post-processing :
  + Term Frequency:
    - tf = 1 / x.sum(axis=1)[:, None]
      * axis=1 means row-wise -> We apply it to each row
      * <https://stackoverflow.com/questions/22149584/what-does-axis-in-pandas-mean>
    - x = x \* tf
      * If there are three words with frequencies 1, 1,1 in a text, it will be 0.3, 0.3, 0.3
    - Makes values more comparable by counting frequencies
  + Inverse Document Frequency
    - idf = np.log(x.shape[0] / (x > 0).sum(0))
    - x = x \* idf
    - Highlights rare words
  + Many variances of tfidf which may work better!
  + sklearn.feature\_extraction.text.TfidfVectorizer
  + N-grams
    - sklearn.feature\_extraction.text.CountVectorizer + Ngram\_range + analyzer

*Text Preprocessing*

* Lower-case
  + CountVectorizer does this by default
* Lemmatisation
  + Same as stemming but with more care
  + democracy, democratic -> democracy
  + see, saw -> see
* Stemming
  + Chops off ending of words
  + democracy, democratic -> democra
  + see, saw -> s
* Stopwords
  + Very common words like articles / prepositions
  + sklearn.feature\_extraction.text.CountVectorizer + max\_df

BOW-Pipeline:

1. Preprocess (Case, Lemmatisation/Stemming, Stopwords)
2. BOW & Ngrams can help to use local context
3. Post-process (TF-IDF)

* Vector Representations
  + Words:
    - Word2Vec, FastText, GloVe
  + Sentences:
    - Combine Word2Vecs via mean, sum
    - Sentence2Vec

BoW vs Word2Vec

* BoW:
  + - Vectors are very large
  + + Vector values are interpretable
* Word2Vec
  + + Vectors relatively small
  + - Seldom interpretable
  + + Words w/ similar meanings have similar vecs

**Images**

* Can be transformed to vectors by taking the output from a layer - 'Descriptors'
  + From early layers: Task-independent
  + From later layers: Task-dependent
* Image Augmentation!

**Exploratory Data Analysis (EDA)**

* Getting comfortable with the task at hand
* May not help sometimes when highly encrypted data -> Models with huge compute might outperform you here
* Gain Domain Knowledge to understand each Data Column
* Check if the data is intuitive (e.g. are age values within what we'd expect)
  + Errors might also be a normal part of the type of data and creating a new column with the feature is\_correct and boolean values might help our model
* Understand the generation of the data
  + Are train & test set generated in the same way? Have the same distribution?
    - If not it's not possible to use part of the train set as the validation set -> Perhaps use an adjusted version of the train set
    - If not given visualize the features & distributions from train & test set

*Anonymized Data*

* Hashed/Encoded text: Won't change anything for BoW, but cannot take pretrained word2vec
* If column names are also encoded, try to guess their meaning & type (categorical, numerical)
* Try to undo scaling such as standard scaling
* df.dtypes / df.info() -> Guesses types for you
  + x.value\_counts() to check for unique values
  + x.isnull() to check for null values

*Visualizations*

* Exploring Individual Features
  + Histograms
    - Splits data into bins - can be misleading hence try a variation of bins
    - Huge spike could be due to missing values filled with e.g. the mean
    - plt.hist
  + Plot index vs value
    - Shows you whether random row order
    - Lots of repeated features?
    - plt.plot(x, '.')
    - plt.scatter(range(len(x)), x, c=y)
      * Adds the labels as color
  + Statistics
    - df.describe()
    - x.mean()
    - x.var()
    - x.value\_counts()
    - x.isnull()
* Exploring Feature Relations
  + Pairs
    - Great to check if distribution in train & test set is the same
      * Even you cannot see the labels via color code, the testing point features should generally be in a similar area on the plot – If not the feature might not have a lot of value or any type of bug
    - plt.scatter(x1, x2)
    - After finding a strong relation it might help to create a new feature based on this
    - pd.scatter\_matrix(df) to get a quick scatter plot for all features
    - Scatterplots don't give great information about densities -> Combine with histograms
  + Groups
    - Look for each features coorelations with all others (Diagonal should be 1)
    - Perhaps custom functions such as how often is this feature larger than the other ...
    - df.corr(), plt.matshow( .. )
    - Try to group features and then generate new features based on the group they belong to
    - k-means clustering
    - Compare all features by their means
      * df.mean().sort\_values().plot(style=' .')
* Try to find a hypothesis & then prove it w/ many visuals

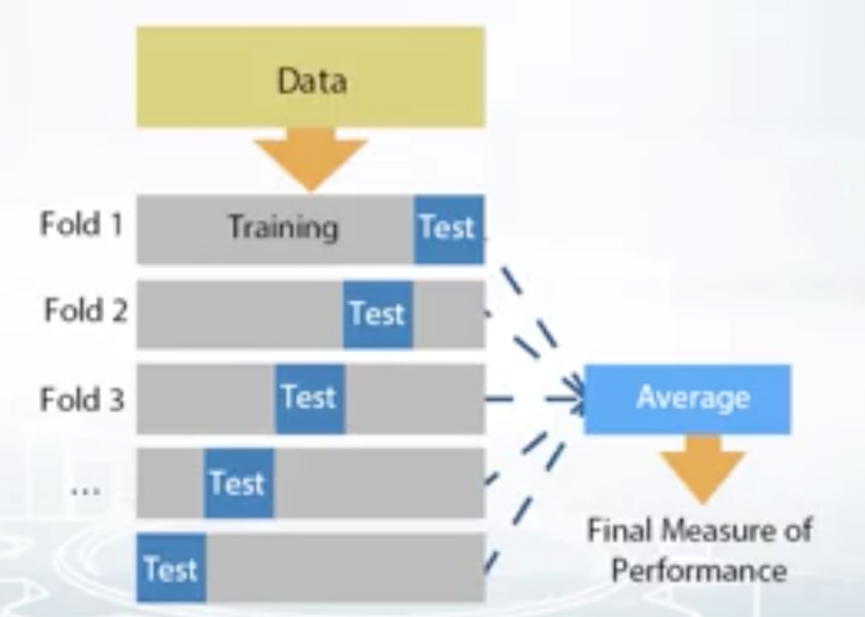
*Dataset Cleaning*

* Duplicated & constant features
  + You might be given e.g. only one year of the dataset and hence one feature, the year, is always constant in your dataset
  + Or a feature constant in the train set, but varying in the test set
    - Drop both, as the model might assign weight to them & they consume unnecessary RAM!
    - df.T.drop\_duplicates()
  + Features might be hidden duplicates, i.e. by replacing them acc. to a dict you get a duplicate feature (e.g. age & year of birth)
    - -> Label Encode them first & then compare
      * Make sure you label encode them correctly (e.g. both by the first value which appears)
    - for f in feature:
      * df[f] = df[f].factorize()
    - Then  df.T.drop\_duplicates()
  + train.nunique(axis=1) == 1
    - & dropna = False, to make sure na is also a unique value and not dropped
* Duplicated Rows
  + Check if rows with same features & label
    - If diff label, perhaps to not give clues & have separate private dataset
  + Try to understand why; It might help you understand their data generation process
* Jitter a value to add Gaussian Noise
  + def jitter(data, stdev):
  + N = len(data)
  + return data + np.random.randn(N) \* stdev
  + # sigma is a given std. dev. for Gaussian distribution
  + plt.scatter(jitter(X, sigma), jitter(Y, sigma), c = y)

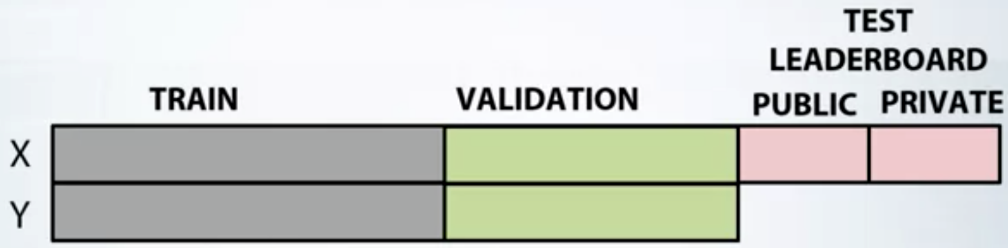
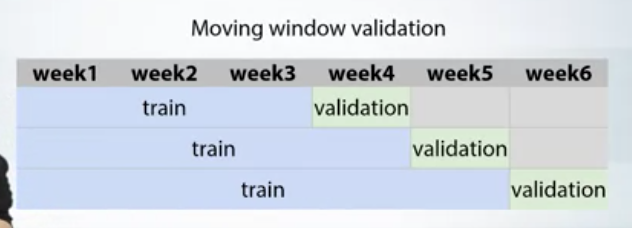
EDA-Springleaf Notebook

* Taking the modular of the an important multiple as a new feature might help -> 12 & 50 in example

*Discrepancies between public & private leaderboard*

* Organizational problems (cannot be influenced)
* Overfitting
  + While in general higher score on train than val set means we are overfitting, in competitions only if our val score is a lot higher than the test score we overfit
* Validation Schemes
  + The following are used for *estimating model quality,* when you submit predictions, in any case train on the WHOLE data
  + Holdout
    - sklearn.model\_selection.ShuffleSplit
    - If the data follows a time-based trend, we should not shuffle it! Else that time-feature will be lost!
      * e.g. Rossman Store Sales competition; Grupo Bimbo , Inventory Demand Competition
    - Samples between train & validation will not overlap (except you have duplicates)
  + k-fold
    - Repeated holdout
    - Will be k-times slower than holdout
    - If all k-folds have roughly the same score we should do holdout, else perhaps average k-scores
    - sklearn.model\_selection.KFold
  + Leave-one out
    - ngroups = len(data) -> Repeating k-fold for every item
    - If we do not have a lot of data
    - sklearn.model\_selection.LeaveOneOut
  + Stratification
    - In case the dataset is very small or unbalanced, make sure to stratify to ensure equal proportions of each class in each set
    - If possible, have the validation set mimic the distribution of the test set!

*Types of splits*

* Random Split
  + There might be dependencies in rows, e.g. if a husband can pay a debt, probably his wife can too -> Generating a feature assigning them as hus & wife might help
* Time-based split
  + E.g. we have one month of sales in the training data and then the week after as testing data -> We might be able to use the weekdays one week prior to the testing week to get good performance
  + Random-splitting is a bad idea here!
  + Moving-Window validation
* Id-based split
  + Random split could miss the belonging of classes
  + E.g. nature conservancy fishing competition -> Photos of fish from one boat; Boats did not overlap in training & test set
    - You should have some boats in validation which you havnt seen before!
    - IDs weren't given, but had to be derived by clustering pictures
* Combined
  + ID & Time - e.g. predictions sales for multiple shops -> ID-based for shops; Time-series for shops sales
  + ID & ID - e.g. Deloitte Housing Competition -> Multiple IDs based on district level

*Problems*

* During Validation
  + Different scores on different folds
    - Too little data
    - Too diverse & inconsistent data
  + Extensive validation may be key
    - Santander Customer Satisfaction Challenge
    - Make k-fold several times with diff random seeds & average scores from different k-fold splits!
    - Separate fold for selecting params & check model quality
* During Submission
  + Leaderboard score consistently higher / lower to val score
    - Check if we overfitted
    - Check if correct train/val split (i.e. try k-fold)
  + Leaderboard score not at  all correlated to val score
    - Submit solutions as early as possible to get a feel
    - Might come from a very different distribution
      * e.g. height of men for test, but height of woman in train data -> Try to shift your data by how much men are taller on avg
    - Leaderboard probing
      * Figure out the difference between train & test data and shift your predictions by that amount
      * E.g. send two constant predictions and calculate the error with their metric used, e.g. MSE
      * E.g. Quora Question Pairs Competition
      * <https://www.kaggle.com/c/quora-question-pairs/discussion/31179>
  + Leaderboard shuffle - Reasons
    - Randomness
      * Often when scores of participants are very close
      * Liberty Mutual Group Property Inspection Comp.
      * Two sigma financial Comp., as highly unpredictable data
    - Too little data
      * Restaurant Revenue Prediction Comp.
    - Big diff in public & private data
      * Rossman Store Sales
        + Time-series based
        + When people adjusted scores to public leaderboard, private leaderboard in fact was the next week after public hence compl. different
  + Choosing submissions
    - Generally you can choose two submissions - it is practice to choose the best performer on your validation set & the best performer on the public leaderboard
    - If the test distribution is very similar to training, your val submission will perform best
    - If the test distribution is very different to training, your leaderboard submission will perform best

*Data Leakages*

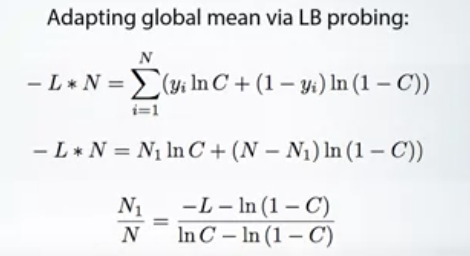
* 'An unexpected information in the data that allows to perform unrealistically good predictions'
* Can only be exploited in competitions (obv. not in real world)
* Check if the dataset is shuffled
  + Plot target value vs row index
    - Smooth with running average
    - Plot rolling\_mean & overall mean target value
      * If the rolling\_mean oscillates around the mean value the data is shuffled
      * If the rolling\_mean at some indexes is very far off from the mean we have some patterns we can exploit

*Types of Data Leakages*

* Time-splits
  + Normally contain some sort of information about the future which can be exploited
  + Check the train/public/private splits
    - -> If the split is not on time, but e.g. random, then you have a leak you can exploit!
    - In such a case, unrealistic features like the price the day after will matter the most
    - -> Even if split on time, we have information about the future
      * We can e.g. get some future features of a user in the test set
      * esp. in Click-Through Rate tasks, Stock-market prediction, Weather, Sales
* Unexpected Information
  + Meta-data
    - e.g. in a dog vs cat classification if one  camera was used for dogs & one for cats -> Pictures might have the same size for each type // the same resolution // other metadata
    - Dato Truly Native Competition
  + ID
    - May be hashed in an secret way e.g. ghsegho
    - Caterpillar Competition - Adding ID as a feature slightly improved results
  + Row Order
    - Sometimes data is shuffled in a simplistic way
    - Telstra Network Competition
  + Other
    - E.g. Row duplication rows next to each other had same value
    - TalkingData Competition
  + -> Try adding Row Number, ID and others to your model as features to probe if results improve
* Only two ways to avoid data leakage:
  + Test set with no features at all, only IDs / no interrelation
  + No future values at all

*Leaderboard Probing*

***Can be used to increase your training set for the final private leaderboard, once you have the perfect score for the public one***

* Categories tightly connected with ID
  + E.g. groups (e.g. by IDs) which always have the same target value -> Even if they are split up between private & public, we can detect them
  + To probe, we set our scores to e.g. all 0 for that particular group (and leave all other groups the same); If the score improves, that's good; if it degrades, we set them all to 1 (i.e. 1 is the ground truth for that group in that case)
  + The category might not necessarily have the exact same value but some other internal relation such as the same mean! - In that case see the formula below (if the evaluation method is logloss!) for calculating the ratio of e.g. 1's in the total test set. (L is the leaderboard score, C is your classification probability (e.g. probe by setting all C's to 0.3) & N is the number of total rows in the public set > Solve for N1
  + Redhat Challenge
  + Expedia: Data contained the distance to the hotel they booked & goal was to predict hotel booking
* Pairwise tasks
  + -> Connectivity Matrix - Incidence & Adjacency Matrices (see notebook), Quora Challenge
* Brute Force
  + Binary Classification: You will need as many tries as there are rows in the test data - You start with all 0's and change one number at a time to 1; With each submission you simply check if your score improved or not
    - For DataScienceBowl 2017, 198 rows & 3 submissions per day, hence 198/3 = 66 days or 198 submissions
* Oleg's approach:
  + <https://www.kaggle.com/olegtrott/the-perfect-score-script>
  + [https://medium.com/@u39kun/i-find-machine-learning-competitions-exciting-and-addicting-438fe95b33f5](https://medium.com/%40u39kun/i-find-machine-learning-competitions-exciting-and-addicting-438fe95b33f5)
  + Possible in 5 days / 14 submissions

Week 1

–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Hardware Requirements:

* RAM – The more data you can keep in memory he better
* Cores – The more cores the faster & more experiments you can do
* Storage – SSD crucial when working w/ images / large datasets

If Laptop:

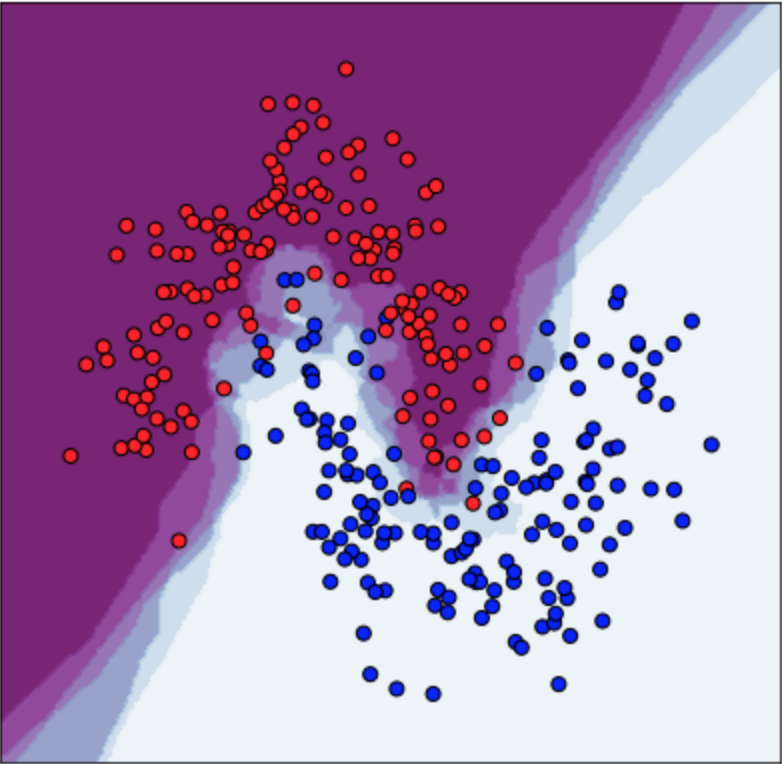
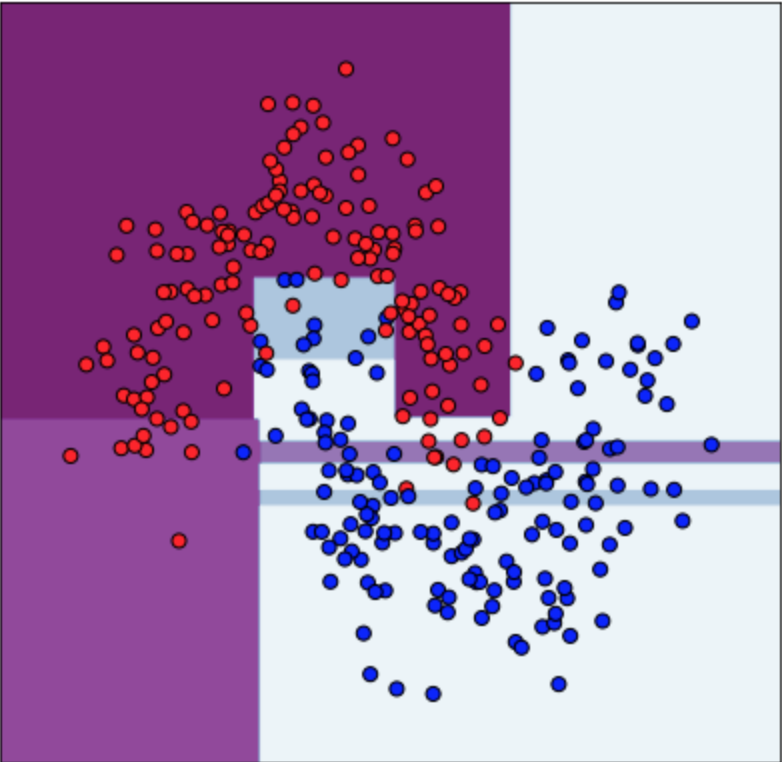
* 16GB Ram, 4 Cores
  + Intel Corei7 has 4 cores

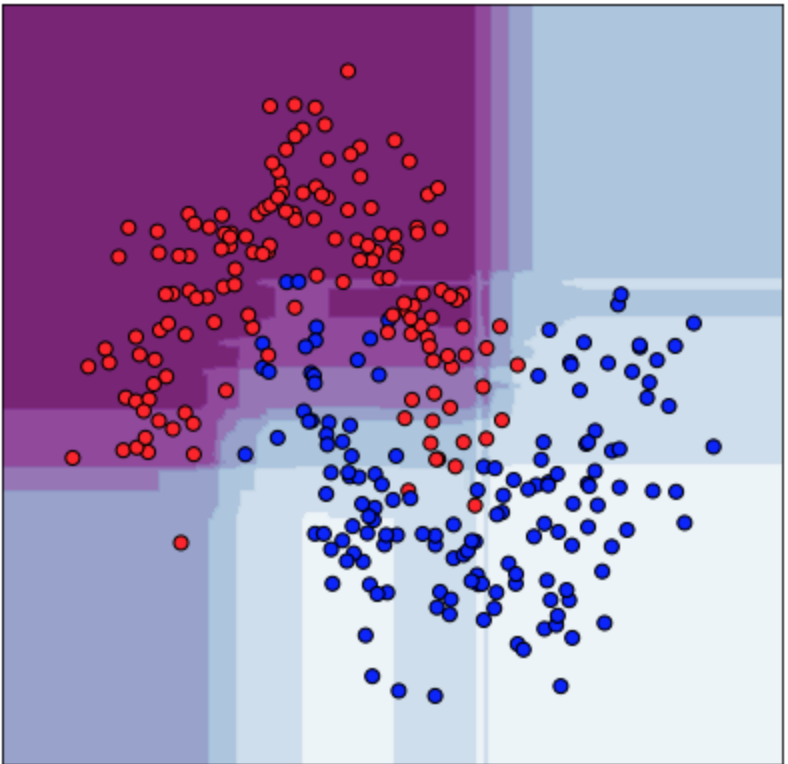
If Desktop:

* 32GB+ RAM, 8+ Cores
  + Intel Core i9 has 8 Cores

Software:

* Python
  + Pandas – Like SQL in Python
  + NumPy
  + scikit learn
  + matplotlib





* Classifiers (L to R): Decision Tree; Random Forest (many DTs); kNN
  + <http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html>
  + <http://arogozhnikov.github.io/2016/04/28/demonstrations-for-ml-courses.html>

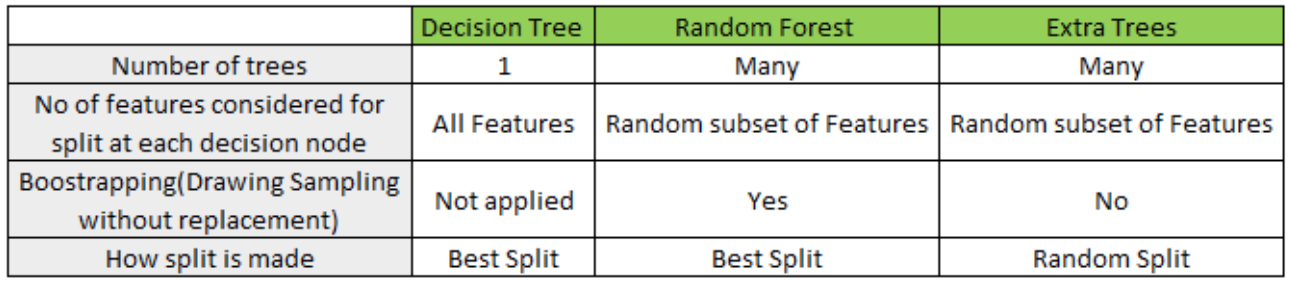
Prevention of overfitting on test set by organizers:

* Limit submissions
* Include irrelevant samples in test set
* Split Score into Public & Private (Public part & Private part of test set – Private determis final score)

Vowpal Wabbit: Scitkit for large datasets

XGBoost: For Grad Boosting

LightGBM: For Grad Boosting

* Linear Models:  
  - Decision Tree: Similar to drawing multiple lines and further splitting into subset
  + - DT’s can be combined in various ways:
    - -> RF, Gradient Boosting DT’s
    - ExtraT vs RF: ET is faster as it tests some random splits over a subset of features, while RF tests all splits over subset of features in order to find the Best
  + GBDT
    - A sequence of trees, each aims to improve predictions of the previous via a learning rate (lower learning rate leads to better generalization, but more computationally exp.; Generally early trees matter a lot more)
  + SVMs
* kNNs:
  + Different distance functions can be configured via scikit (Example of kNN on PIC)
* NNs:
* No Free Lunch Theorem:
  + There’s no fit for all algorithm
* GBDT & NNs are normally the strongest
* Bootstrapping:
  + ‘A self-starting process’