How to Win a Data Science Competition: Lean From Top Kagglers

Coursera - National Research University Higher School of Economics

2020

Chapter 1

week1

1.1 Introduction and Recap

- In Kaggle competitions, looks for **Kernels** and/or **Notebooks** - people share code that way and you can learn from them.
- IMPORTANT: Insight is more important than algorithm
- IMPORTANT: Don't limit yourself - use advanced feature enginnering; run huge greedy calculations over night.
- Recap of models/methods:
  - Linear Methods
    - Logistic Regression; Support Vector Machine
    - GOOD FOR: Sparse, high dimensional data
    - BAD: Often linear separability is not a good approximation
  - Tree Based Methods
    - Decision Tree; Random Forrest; Gradient Boosted Decision Trees (GBDT)
    - GOOD FOR: good default method for tabular data; good for non-linear relationships
    - BAD FOR: hard to capture linear separation (say diagonal line in a 2D plane)
      from sklearn: can create over-complex trees that do not generalise data well - that is they are prone to overfitting
      trees can't extrapolate trends - they are good at interpolation not extrapolation
    - *Sklearn* has good random forrest; *xgboost* has good GBDT
  - K-Nearest neighboors (KNN)
    good for feature building
  - Neural Networks (NN)
    good for images, sounds, texts and sequences
    very good framework is *pytorch*
- Most powerfull methods currently are GBDS and NN
- No free lunch theorem - there is no single methods that beats all other methods for all applications
- TODO: Look into H20 - open source, in-memory, distributed, fast and scalable machine learning and predictive analytics platform.
  Looks quite interesting!
- Hardware: SSD is critical; use cloud computing for anything too big.

1.2 Feature Preprocessing

- VIP: Strong connection between preprocessing and model choice
- Scaling: $x_i \to \alpha x_i$
  - decision trees are not affected by a scaling tx, but non-tree based methods like NN, kNN and regularized linear models are very affected
    in KNN - $x \to 0 * x$ means ignore x, while $x \to \infty x$ means x dominates
    Any methods that relies on gradient descent is sensitive to scaling (logistic regression, SVM, neural networks, etc. etc)

- appraoch 1: Normalization (map to [0,1])

$$x \to (x - x.min())/(x.max() - x.min)$$

  code: sklearn.preprocessing.MinMaxSaler
- approach 2: standardization (to $\mu = 0, \sigma = 1$)

$$x \to (x - x.mean())/x.std()$$

  code: sklearn.preprocessing.StandardScaler
- Can use scaling to boost or attenuate signals - so scaling can be thought as just another hyperparameter to tune

  In general, start with all features in the same scale and explore changing that.

- Outliers
  - linear models extremely sensitive
  - approach 1: **Winsorization** - clip values between percentiles

    code: UPPER, LOWER = np.percentile(x, [1,99]); y = np.clip(x, UPPER, LOWER)

    very common in finance
  - approach 3: Rank

    good for knn and linear models when you don't have time to handle outliers by hand
  - approach 3: Other transformation

    log transformation; raising to power ¡ 1 - these have the benefit of bringing outliers in closer, and spreading things around zero apart

- Missing values
  - can be Nans, empty strings, outliers like -1, or -999
  - identifiying when a value actually represents a missing value can be challenging. Main tool: Histogram.
  - Sometimes missing values contain alot of useful info - there might be a reason the value is missing

    So it is good to create new features like 'isMissing' or something like that.
  - 3 main imputation techniques
    * NaN → value outside range

      Gives trees possibilities to use this; Performance for linear models suffers greatly

      Curated data often comes with somethign like this already (as described above)
    * NaN → mean, median or mode.

      good for linear methods; not good for trees
    * Try to reconstruct NaN

      not easy; need to know something about the data generating process.
  - WARNING: be vary careful with early NaN imputation if you then build features based on imputed feature

    for exampl if you fill a cyclic feature with median values, and then construct diff as new feature - will lead to prominent discontinuities and flat zones.
  - some libraries, like XGBoost can handle NaNs automatically

- For all all of these, you must learn the transformation from training data, and then apply the same one on testing. Sklearn Transformation API (used by the transformations above) allows you to do this. VERY GOOD.
- USEFUL TRICK: create different predictors from same feature using different preprocessing techniques.
- USEFUL TRICK: Mix models that are trained on different preprocessed versions of the data

1.3 External reading

- FROM SKLEARN
  - when data is sparse, you don't want a scaler that moves zeros. Think about nature of your data to chose scalar.
  - Transfromer API: scaler = sklearn.Preprocessing.StandardScaler().fit(X_train); x_test_scaled = scaler(X_test)
  - many estimators in sklearn expect features to look more or less $\sim$ N(0,1)
  - VIP: Note l1 and l2 regularizes assume all features centered around zero and have variance in the same order.

  - other useful tx:
    * preprocessing.Normalizer() - scaling individual samples to have unit norm (useful when estimating pair similarity via dot prod)
    * preprocessing.OneHotEncoder(), preprocessing.OrdinalEncoder()
    * preprocessing.KBinDiscretizer(n_bins, encode).fit() - different strategies available
      histograms focus on counting features in particular bin, whereas dsicretizers focus on labeling features with bin

- Sebastian Raschka
  - Tree based classification is probably the only scale invariant algo
  - VIP: when using PCA, it is better to standardize (mean 1, std 0) than just normalize (map to [0,1]) - scaling affects covariance
    cool example doing PCA and then bayes classifier on top - compar prediction score.

- Jason Brownlee - Discover feature engineering: "Most algorithms are standard - we spend most of our efforts on feature engineering"

- Rahul Agarwal - Best practices in feature engineering
  - LogLoss clipping technique: clip prediction prob to [0.05. 0.95] when using log loss metric - it penalizes heavily being very certain and wrong
  - Use PCA for rotation, not only dim rec
  - sometimes add interaction features, A*B, A/B, A+B, A - B

1.4 Feature Generation

- encodings categorical/ordinal
  - label encoding: map categories to 1,2,3,4...
    good for tree based methods, not good for knn or linear (because it assume order and proportionality in labels - moreover dependence is likely not-linear)
    code: sklearn.preprocessing.LabelEncoder() or pd.factorize()
  - Frequency encoding: map to numerical value representing frquency of category
    can help trees use less splits
    this is even sometimes useful in linear models
  - one-hot encoding: create indicator variable for each category
    these can be good for linear methods, but in general slow down methods (explosion of features) and might not help (specially trees)
    though will help tree if target depends on lbael encoded feature in a very non linear way
    code: pd.get_dummies() or sklearn.preprocessing.OneHotEncoder()

- Datetime and coordinates
  - Very different than simple numerical or categorical because we can interpret their meaning - they have much context
  - dates and times can lead to two main types of features
    * moments in a period (ie using periodicity of datetime)
      ex: day of week, day of month, month in year, etc. Or minute value, hour value, etc.
      useful to capture repetitive pattern in data
    * time since/to event
      can be row independent: years since 2000, for all rows (so all rows have same reference)
      row dependent: days since last holiday (or till next holiday) - two rows can be referring to different holidays
  - very useful to diffs between two date columns
  - once you generate new features, numerical or categorical, preprocess them accordingly
  - coordinates
    * typically you want to calculate distance to points of interest (nearest hospital, school, etc)
    * very useful to calculate aggregated statistics for objects around an area
      ex: # of flats around a point -¿ proxy for popularity of area
      ex: mean price of flats around a point -¿ gives sense of price of area.

- Collection of tricks
  - separte price into integer part and fractional part - can utilize differences in peoples *perception* of a price
  - Create feature, 'isRoundNumber' - people often use numers like 5 and 10, while robots can use many decimals.
  - One-hot encode interaction between categorical features [just concatenate strings and OHE result]
    Not so useful in tree based models, because they can easily approximate this with individual categories.
  - Sometimes simple multiplication, or division of features makes a huge difference.
    linear models can't approximate these, and trees have a very hard time approximating them.
  - sometimes useful to add slightly rotated coordinate system - particularly when using trees.
    ex: if a particular street happens to be a good division, but that street is not algined with coordinates, then tree uses many split to approximate.
    hard to know what rotation to use a priori - so add several and check effect.

1.5 Feature Extraction From Texts and Images

- For Text...
- Preprocessing: 1) lowercase, 2) lemmatization, 3) stemming, 4) remove stopwords
- feature extraction: Bag of words - column per word in corpus, row perd doc, count ocurrences
  can extend to n-grams, of either words of letters
- Post processing: TFiDF (Term Frequency and Inverse document frequency)
- OR ...
- embeddings (word2vec, or others)
  - Still use preprocessing
  - create vector representation of words in text
  - uses nearby words (unsupervised)
  - often resulting vector space has interpretable operations
  - training can take a long time - check for pretrained
- BOW and Word2Vec often give very different results - use bott together
- for images
  - look for pretained models and do some fine tunning
  - use image augmentation to increase training samples (crops, rotations, adding noise, etc.)
    reduces overfitting

1.6 Questions

- in GBM_drop_tree notebook - why are raw predictions - the output of the staged_decision_function - approaching ± 8? (I assume it has to do with depth 3 choice of trees, but still, shouldn't output be close to ± 1 which is actual y-values?

Chapter 2

Week 2

2.1 EDA

- VIP: **EDA is key**

  Kaggle CEO says two apporaches: 1) is EDA other is 2) deep NN and pass everything. They have different domains in which they work better.

- Steps:

  1. Get Domain knowledge - google, read wiki, read state of the art (put in the time!)
  2. Check whether values in data set agree with domain knowloedge - var ranges, typos, etc. (systematic errors can be very useful info)
  3. KEY: figure out the generation process used in the data - must try to mimic to set up proper validation scheme.

     perhaps they did random sample; perhaps they oversampled a class to balance out; some times train and test set are produce very differently. INVESTIGATE

- Exploring Anonimized data
  - anonimized data is data which orgranizers intentionally change so as to not reveal some information (ex: replace words with hash values, or col names with x1, x2 ..)
  - Things to try:
    * try to decode (very diffiicult, and most of the time, can't do it)
    * guess the meaning of the column

      cool example was trying to unstardadize a numerical column, by reverse engineering the mean and std dev, to find out original column was year born
    * guess the type: numeric, categorical, etc. - this is generally doable
    * find out how feature relate to each other - find pairwise relationships (scatter plots) or even groups (correlatio plots, etc)
  - tip: label encode using pandas factorize (encodes based on order of appearance)

    `for c in train.columns[train.dtypes == 'object']: x[c] = x[c].factorize()[0]`
  - **VIP TIP**: fit a random forrest and then use `plt.plot(rf.feature_importances_` - this can tell you which features you should work on most.

- VIP: Linear models can eaily extract sums and differences, but tree based methods cannot!!

2.2 visualizations

- EDA is an art, and visualizations are our tools
- Plots to explore individual features
- VIP: Never make a conlusion from a single plot! if you have a hypothesis, try to come up with several different plots that could disprove it!
  - `plt.hist`

    can be misleading - vary bin nums. Also zoom in.

    if see alot of thinks like 12, 24, 26 - i.e. separated by same amount then generate feature x % 12 (or whatever the appropriate number)

- plt.plot(x,'.')

  convenient not to connect points with line segments - just use dots

  if horizontal lines appear, then lots of repeated values - IN THIS CASE, CREATE FEATURE THAT COUNTS HOW MANY COLS SAME IN THE GROUP - or feature for more nuanced pattern in group.

  if vertical patterns, then data is not shuffled - IN THIS CASE ADD ROW INDEX AS FEATURE

- plt.scatter(range(len(x), x, c=y) - very good for looking for separation in the classes based on that feature.

- Explore feature relationships

  - plt.scatter(x1, x1) - One of the best tools!

    usually color by class label too

    for regression used heaptmap type colors, or visualize target value as point size

    TIP: useful to overlap colored train data with uncolored test data

    scatter plots can lead to finding mathematical relations between features (like $x1 \leq x2$) - use these to generate new features like diff or ratio

    if small number of features: pd.scatter_matrix(df)

  - large scale feature similarity: df.corr() and plt.matshow()

    instead of corr, try to create matrices like: how many times in one feature greater than the other (this can spot cummulative features), or how many distinct combinations of these two features in data exist (can spot relationships between labels)

    really cool algorithm for grouping based on these corr values: **spectral biclustering algorithm** - LOOK INTO THIS!

  - df.mean().plot(style='.')

    particularly if you sort the columns based on that statistic - might see groups

  - AND MANY MORE - BE CREATIVE

- if you find groups of related features, it is often good to generate new features like a mean value of the group, etc.

- nice code when overlapping values:

```
def jitter(data, stdev):
    N = len(data)
    return data + np.random.rand(N)*stdev
plt.scatter(jitter(x, sigma), jitter(y, sigma), c=y)
```

2.3 Dataset cleaning and things to check

- Find (and discard) features that are constant in train and test

  traintest.nunique(axis =1) == 1

  if constant in training and non-constant in test, still remove

- find and remove duplicate features

  traintest.T.drop_duplicates()

  if duplicate categorical but with diff cat names label encode first, using factorize()!!

```
for f in categorical_feats:
    train[f] = traintest[f].factorize()
traintest.T.drop_duplicates()
```

- check for duplicate rows

  check if duplicate rows have diff targets - if so, understand why!

- check if train and test have common rows - if so, why?

- check if data is shuffled (plot feature or target vs row index)

  if not shuffled, high chace leakage exists

- Nice code - feature histograms

```
def hist_it(feat):
    plt.figure(figsize=(16,4))
    feat[Y == 0].hist(bins=range(int(feat.min()), int(feat.max()+2)), normed=True, alpha=0.8)
    feat[Y == 1].hist(bins=range(int(feat.min()), int(feat.max()+2)), normed=True, alpha=0.8)
    plt.ylim((0,1))
```

- Nice code - compare categoricals

```
train_enc = pd.DataFrame(index = train.index)
for col in tqdm_notebook(train.columns):
    train_enc[col] = train[col].factorize()[0]

dup_cols = { } for i, c1 in enumerate(tqdm_notebook(train_enc.columns)):
    for c2 in train_enc.columns[i+1: ]:
        if c2 not in dup_cols and np.all(train_enc[c1] == train_enc[c2]):          dup_cols[c2] == c1
```

- Usually convenient to concatenate train and test, and do all feature engineering with result (but not always)

2.4 Validation

- Validation is a piece of test data not used for fitting, but rather checking the value of the model over unseen data.
- Overffiting in general != Overfitting in competitions

  former: train quality ¿ test quality

  latter: when quality in test is worse than expected
- MAIN POINT: have train/validation split mimic the train/test split - THIS IS KEY.

  without this, you validation score does not represent your out of sample test score.

  Sometimes it can be very challenging to figure out how train/test split was made. It is worth spending considerable time here if needed.
- validation strategies
  - 3 main: Holdout, K-fold, and leave one out.
  - holdout - usually a good choice when there is enough data

    no overlap between train and validation
  - k-fold - basically repeated holdout, where entire data is partitioned into k validation sets - for each validation set, model is trained on complement. Final measure of performance is average over k folds.

    core idea: every sample is used for validation exactly once

    usually k=5 is a good starting point
  - LOO - is k-fold where k = len(train)

    used only when there is very little data
- usually houldout and k-fold on shuffled data not good when:
  - unbalanced data sets (as far as classes)
  - multiclass classification with many classes
- in these cases use **stratification** - preserve the same target distribution in the different folds
- these methods are also generally not good for time series data
  - you generally need to make a time split - so if holdout, don't shuffle.

    this emulates how time series data is received, and used
  - the time series version of k-fold, is a *moving window cross validation* - this rely on capturing trends
  - OTOH, if for competiotn, test/train split did not use a time split, that means you have future data. Then USE it! And have your cv emulate (this is generally not the case)
  - models/features that rely on trends tend to be very different than those that rely on future data, so it is key to get this right.
- main types of splits used in competitions
  - random split (rowwise) - done when rows are fairly independent of each other
  - time based split
  - by ID - (typically several rows per ID)

    test will have IDs not seen in train
  - combined - ex: date split for each ID independently
  - non-trivial
- Typical problems encountered during validation

  - Different folds lead to very different values

    can happen when different folds are very different in nature: example, in competition, cross-validating with january vs february can be very different (number of holidays is very different) HINT HINT

    can also be caused by too little data, or data that is too diverse, or data that is inconsistent (i.e. similar samples with very different target values - error changes if they are both in train, or one and one)
  - **leaderboard shuffle** - very different public/ private scores.
  - some solutions: increase k. Or redo k-fold with different seeds (can use one to get parameters and one to test)

- submission stage problems: LB consistently higher (lower) than cv; LB uncorrelated with CV

  can be caused by train/test data comminb from diff distributions – Best friend EDA: problem is typically that you haven't mimicked split correctly

  Other trick: adjust solution based on LB - find optimal constant to add/subtract from your preductions based on public leaderboard score

  most often problem comes from imbalanced classes. solution, try to mimic the split.

  can also be casued by too little public test data; in this case just trust you CV

2.5 Data leakages

- very bad, people get very sensitive, to exploit or not exploit? (shouldn't and often can't in real world)

  When they exist, they tend to dominate

- typical leaks
  - future peaking [incorrect splits]
  - metadata
  - IDS sometimes contain information
  - row ordering
  - LB probing (can be used to extract what they call *ground truth*

2.6 Additional Material And Links

- CV in sklearn
  - Validation is a way to fit hyperparameters of model without burning data
  - KEY: Preprocessing should also be learned from training, so it is good to use a **pipeline** to combine prepreocessing with validation

    ex:
    ```
    from sklearn.pipeline import make_pipeline
    clf = make_pipeline(preprocessing.standardScaler(), svm.svc(c=1))
    cross_val_score(clf, X, y, cv=cv)
    ```
  - random split: sklearn.model_selection.train_test_split
  - k-fold: sklearn.model_selection.cross_val_score(df, x, y, cv=5, scoring='...')

    can pass a cross-validation scheme to cv to have more control, or even a custom iterable yielding (test, train) splits

    more flexible variant: cross_validate
  - many cross validation iterators that can be used, for iid and not iid (grouped, imbalanced classes, time series)

- Some dude's lessons
  - always cv (not just v)
  - trust good CV method more than LB
  - for final submission picke two very different models (ie. from bagging of SVM, vs Random forrest, vs neural network, vs linear models)

- Kaggle CEO lessons learned
  - Use sklearn pipelines to avoid leakages, peaking, etc.
  - in 2011 random forrest won alot; in 2012 deep NN started dominating (at least vision and time series); in 2015 XGboost started dominating
  - Two main approaches to dominate

    * XGBoost with serious EDA
    * deep NN with very little EDA
- Top Kaggle participant attributes
    * creativity - create lots and lots of features
    * Tenacity - keep working, not get despirited
    * very good with statistics [avoid overfitting]
    * good software practices [like VC]
- suggestion: look at kaggle public data sets, and use their script forking

Chapter 3

Week 3

3.1 Evaluation Metrics

- Problem: Some metrics cannot be optimized efficiently. So need to come up with proxy metric, and find heuristics to transform optimized metric results to final submission.
- also, if train and test sets are different, might need to modify optimization metric a bit (happens often in time series, where distribution changes over time)
- interesting example metrix (comes from finance)

$$Loss(\hat{y}_i, y_i) = \begin{cases} |y_i - \hat{y}_i| & \text{if trend predicted correctly} \\ (y_i - \hat{y}_i)^2 & \text{if trend predicted incorrecly} \end{cases} \tag{3.1}$$

  punishes one type of error much more.

  Hard to optimize with an algorithm

  in his case, he hacked the metric - it was all about making small predictions, and guessing right size (or something like that)

- Do exploratory analysis of unusual metrics!
- ALWAYS BUILD A BEST CONSTANT BASELINE (see below)

3.1.1 Regression metrics

- MSE := Mean Squared Error

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3.2}$$

  optimal constant is mean value of target

- RMSE := square root of MSE

  square root is to make units of error same as target

  every MSE minimizer is a RMSE minimizer and vice versa

  slight differentce in gradient descent, because

$$\frac{\partial RMSE}{\partial \hat{y}_i} = \frac{1}{2\sqrt{MSE}}\frac{\partial MSE}{\partial \hat{y}_i}$$

- R-squared

$$R^2 := 1 - \frac{MSE}{Var(Y)} \tag{3.3}$$

  where in this case Var is the biased 1/N type.

  puts error on reasonable scale: 0 if prediction is no better than optimal constant (negative if worse) and 1 if prediction is perfect

  to optimize R^2, just optimize MSE - just off by two constants

- MAE := Mean Absolute Error

$$MAE := \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{3.4}$$

  more robust than MSE (less penalty on extremes).

  widely used in finance (of by \$10 is exactly two times worse than off by \$5)

  optimal constant is median

  choice on using MSE vs MAE depends on whether outliers data are bad data, or real data that is just rare.

- Problem: 9 vs 10, is same error as 999 vs 1000 in all of the above - this leads to MSPE and MAPE
- MSPE (:= mean squared percentage error)

$$MSPE = \frac{100}{N}\sum_{i=1}^{N}\left(\frac{y_i - \hat{y}_i}{y_i}\right)^2 \tag{3.5}$$

- MAPE (:= Mean absolute percentage error)

  same but with L1 instead of L2.

  These can be thought of weighted versions of MSE and MAE (with weights not summing to one)

  optimal constants are weighted versions of mean and median.

- RMSLE (:= Root mean square logarithmic error)

  basically RMSE where you transform $y_i \to \ln(y_i + 1)$ (and same for hat version)

  this one cares more about relative errors

  error is assymetric, always better to be above than below

  best constant is obtained by going to logspace, getting mean, and transforming back

3.1.2 Classification Metrics

- notation: N is number of objects, L is number of classes, [a = b] is indicator function, 1 if arg is true, 0 otherwise.
- Disitinguish between: **hard prediction** - the actual predicted category, vs **soft prediciotns** - the probability for each category
- metric 1: Accuray

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N}[\hat{y}_i = y_i] \tag{3.6}$$

  uses only hard predictions

  best constant is most frequent class

  bad metric with very unbalanced classes, bc best constant gets you really good score

  difficult to optimize

  doesn't care about confidence, just right or wrong.

- LogLoss

  for binary (1s, and 0s, specifically)

$$LogLoss = -\frac{1}{N}\sum_{i=1}^{N} y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i) \tag{3.7}$$

  Multiclass generalization

$$LogLoss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{l=1}^{L} y_{il}\ln(\hat{y}_{il}) \tag{3.8}$$

  in practice predictions clipped to avoid nans: $\min(\max(\hat{y}, 10^{-15}), 1 - 10^{-15})$

  Logloss penalizes severe mistakes!

  best constanc for log loss is set α_i (prob class i) to frequency of ith class in target

- AUC ROC (Aread under curve - receiver operating curve)

  only for binary tasks

  doesn't care about threshold, just ordering of predictions

  HIS FAVORITE ONE - READ MORE ABOUT THIS!

- Cohen's Kappa

  basically rescales accuracy so that 0 is not 0 accuracy but instead the baseline prediction.

  read more about this.

3.2 Metric Optimization

- what we want to optimize is not always what the model optimizes

  some metrics can be optimized directly by the libraries available: MSE, LogLoss, MAE, etc

  sometimes can use an equivalent loss: for example MSPE, RMSLE, can optimize MSE instead if done correctly

  someitmes have to use a different loss, and simply postprocess predictions and apply heuristics to adjust for target metric

- One method that always works: **early stopping**

  train using one metric, and validate using another. Then stop training when validation on second metric is lowest.

- some tricks:
  - often cant optimise MSPE and MAPE out of the box; but you are allowed to pass sample weights. So you can recreate it.
  - another method for above is to resample dataset according to said weights

    leave test set as it. Usually resample many times and average.
  - for RMSLE just transform to log space, optimize MSE and then transforom predictions back
  - for classification - you often have to callibrate prediction; basically you fit another model to transform scale predictions appropriately (something simple like a linear model)
  - Sometimes just tune threshold using grid search

3.3 extra readings

- All about Learning to Rank problem and some libraries that deal with it
- read more when needed.

3.4 Mean encoding

- main concept: use target to generate new features

  This often leads to more separable features, which leads to trees needing less depth to reach similar predicitive ability

  the more non-linear the target dependency on the feature, the more effective mean encoding can be.

  TIP: If increasing tree depth makes both in sample and out sample better, this is tell-tale sign that mean encoding is probably useful (some features probably have a tremendous number of important split points)

- aka likelihood encoding aka target encoding
- simplest case: given a categorical variable, encode each level of that variable using the correspoinding target mean for that level
- many pitfalls when it comes to doing this - can lead to leakage and overfitting

  VIP: always train encoding on train data only!

- approaches:
  - likelihood = (ones/(ones + zeros)) = mean(target)
  - weighted evidence = $\ln(ones/zeros) * 100$
  - count = ones = sum(target)

- diff = ones - zeros

- regularization for encodings - 4 main approaches
  - cross validation loop (robust)
    split into k-folds (usually 4 or 5) and estimate encoding for values in each fold using only complement.
    How you train this in train, and then use in validation/test? Do you average the maps from the folds? Do you treat it as a new fold?
    Careful - in extreme example, folds = num data, can end up with perfect leak of data
  - Smoothing
    $$\frac{mean(target) * nrows + alpha * globalmean}{nrows + alpha}$$
    where nrows is nrows for that given category value, globalmean is the mean of target accross all category values, and alpha is a fixed constant that determines the strength of the smoothing.
    This punishes rare category encoding.
  - Noise - just add random noise to target values before doing straight encoding
    degrades quality of encoding on train data.
    pretty unstable; must be tunes very carefully
  - Expanding Mean
    sort data. Then use only rows from 0 to n-1 to calculate encoding for row n
    How do we sort the data? - straightforward when time series, but what if not?
    this one introduces the least amount of leakage and requires no hyperparameter tuning - it is his favorite.

- Extensions and generalizations
  - in binary classification, mean is the only relevant statistic. For regression can use things like median, percentiels, std, bins, etc.
  - in multiclass classification, every feature with lead to L different encodings, one per class.
    models often solve mutliclass in one-vs-all approach, so these encodings are a good way to give model additional info about other classes
  - many to many relations - one approach is to stack data, encode results, and then unstack and assign vector value to original, then convert that vector to a useful stat like mean, or std.
    example: col1 - USERID, col2 - list of apps on phone, target - 0 or 1. Approach is to stack data, have one col for each User/app pair. Encode Apps, and then stack again - will lead to a vector of encoded values (apps) per user.
  - time series
    can create very useful and complicate features - i.e. rolling statistics of target variables
    example: for a given category, calculate mean from previous day, two days, previous week, etc.
    also, can aggregate along different features. example: col1 Day, col2 user, col3, expense category, col4 amount spent - can do per user average of prev day spending; can also do per category spending for previous day, etc.
    VIP: don't use future data
  - numerical features
    for numerical features can bin and treat as categorical
    one approach to binning is to fit tree on raw data, and encode based on tree splits
    if a feature has many splits, it is a good idea to bin (probably using splits of tree)
  - Interactions
    simialrly fit tree on raw
    in resulting trees, look at neighboring splits (i.e. plot tree, and look at all neighboring nodes). The pairs that appear most often, probably have meaningful interaction.
    then simply concatenate and encode as usual
    LOOK INTO cat_boost

3.5 From Programming Excercise

- cool code: transform
  `all_data['item_target_encode'] = all_data.groupby('item_id')['target'].transform('mean')`
  transform returns data frame with index like original df (using the mapping created by the aggregation)
- For competition: Expanding mean scheme - led to highest correlation of feat with target
- For competition: preprocessing approach - for each month, they get all shops and all items, and create the cross product - and fill out with zeros any pair that doesn't have data