Model assessment

How good is my model and how do I properly test it?

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Terminology: Loss function vs metric

- *Loss function*: minimized to train a model (if appropriate)
  E.g., gradient descent uses loss to train regularize linear model
- *Metric*: evaluate accuracy of predictions compared to known results (the business perspective)
- Both are functions of y and $\hat{y}$, but loss is also possibly model parameters (e.g., linear model regularization loss tests parameters)
- Examples:
  - Train: MSE loss & Test: MSE metric
  - Train: MSE loss & Test: MAE metric
  - Train: Gini impurity & Test: misclassification or FP/FN metric
- If metric is applied to validation or test set, informs on generality and quality of your model

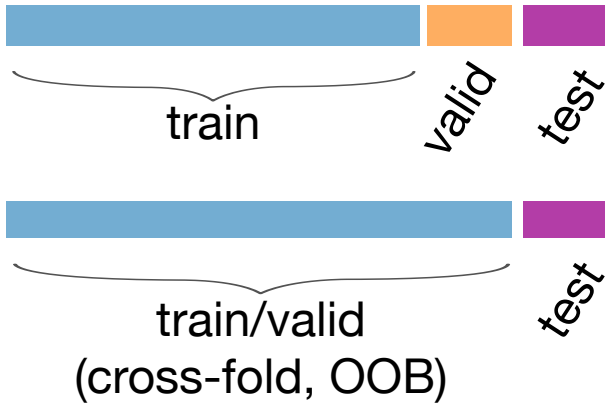
See also stackoverflow post by Chstiros Tsatsoulis: https://goo.gl/T5AmrT

Train, validate, test

- *This might be the most important slide of entire class!*
- We always need 3 data sets with known answers:
  - training
  - validation (as shorthand you'll hear me / others call this test set)
  - testing (put in a vault and **don't peek**!!)
- Validation set: used to evaluate, tune models and features
  - Any changes you make to model tailor it to this specific validation set
- Test set: used exactly **once** after you think you have best model
  - The only true measure of model's generality, how it'll perform in production
  - Never use test set to tune model
- Production: recombine all sets back into a big training set again, retrain model but don't change it according to test set metrics

How to extract validation, test sets

- Extract random subsets; perhaps 70%/15%/15%; can shuffle then chop
- Or, grab 15% test set (and hide it away) & use OOB or cross-fold on remainder for train/valid
- For RF, we can start with out-of-bag score
- Ensure validation set has same properties as test set (e.g., size, time, …):
  - if 10k samples in test, make 10k sample validation set
  - if test set is latest 2 months, validation must be latest 2 months of remaining data



Key question: is your data time-sensitive?

- Time series sometimes obvious: temperature, stock prices, sales, inflation, city population, …
- You can try to detrend the data to flatten average *y* etc…
- Almost all data sets are time sensitive in some way (boo!)
- Some data sets are skewed over time even if no date column; e.g., new users to facebook are different over time
- Try to find things that are less time dependent; e.g., air conditioning sales appear to fluctuate over time but these sales are driven more by associated temperature and humidity than date
- Create lagging windows; e.g., average sales per day over the last week
- Try to convert absolute dates (and date-related variables) to relative variables, such as **age** computed from **salesdate** - **manufacturingdate**

Splitting time-sensitive data sets

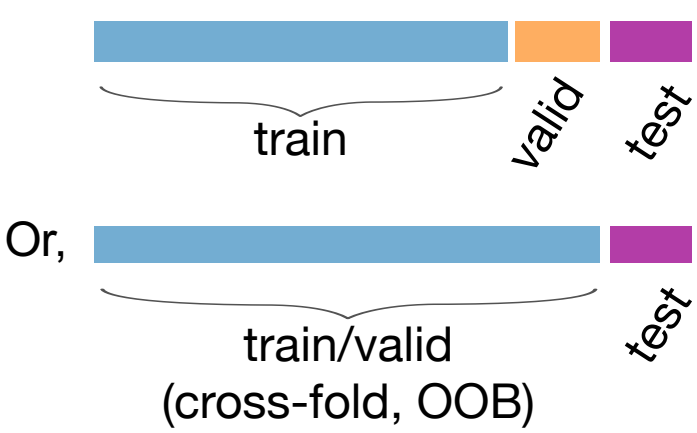
- If your data set is time-sensitive, do not shuffle: sort by date then use newest rows as valid/test sets
- This means OOB cannot be used for time-sensitive data sets as it randomly selects test records

```
df = df.sort_values('saledate')
n_train = len(df)-n_valid
df_train = df[:n_train]
df_valid = df[n_train:]
```

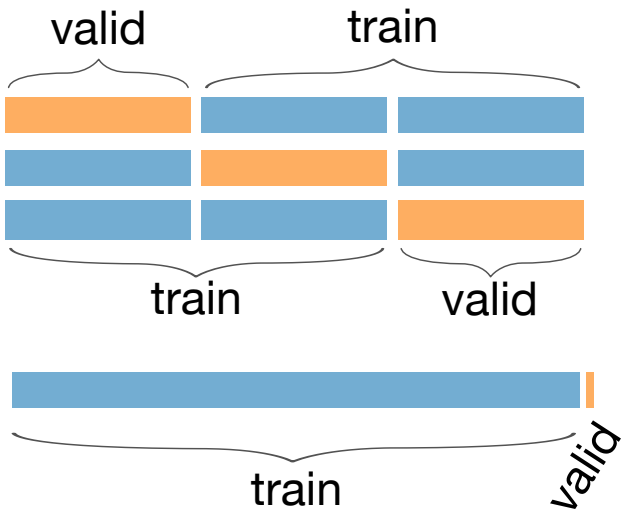
	SalesID	saledate	Hours	UsageBand	Group
Training	1007352	2/19/08	1171	Low	TEX
	2297737	8/15/08	5659	Medium	TEX
	2306055	9/19/08	8999	Medium	TEX
	2275176	10/16/08	4425	Medium	SSL
	2274051	12/16/08	2425	Low	TTT
	2268394	1/31/09	3112		WL
	2288319	2/9/09	5089	Low	BL
	1745722	2/17/09	4489	Low	BL
	2297316	2/27/09	963		WL
	1896854	3/12/09	2851	Medium	BL
	2298201	11/14/09	5075	Low	BL
	2298929	2/15/10	13173	Medium	TTT
Validation	2276590	3/4/10	6758		TEX
	2298928	5/6/10		Medium	WL
	2277691	2/3/11	7191	Low	MG
Testing	2296994	2/10/11	3783		SSL
	2294960	3/18/11	8201	Low	
	2288988	7/27/11	9115	Medium	WL

Testing strategies

Always split out test set



Validation cross-fold or leave-one-out



Metrics interpretation

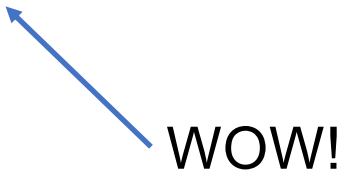
- Basic idea: for each test record, compute error from y and $\hat{y}$; the metric is then usually the average of these errors
- **Perspective**: Is 99% vs 99.5% accuracy difference 2x or 0.5%? What about 80% vs 90%? 10% diff but also 2x
- As we approach 100%, getting better is tougher and tougher
- Is 90% accuracy or R^2=0.8 good? Maybe. What are the lower bounds from a simpler or trivial mode?
- Classifiers must beat *a priori* probabilities
  - If 90% of email are spam, your model must beat 90% accuracy
- Regressors should beat "mean model" and linear model

Training score

- Training score not really useful by itself because, for example, we can get good fit for random data X->y with 1000 x 4 data X data, R^2=.85
- Actually, if training score is low, model is too weak
- Or, dataset is missing exogenous vars like “we had a sale that day” or “closed on holidays”

```
X_train = np.random.random((1000,4))
y_train = np.random.random(1000)
rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train, y_train)
rf.score(X_train, y_train)
```

```
0.847473179728131 4
```



What if training score is good but validation is very low?

- Overfit model?
- Bad validation set?
  (didn't extract random or time-sorted set or just given bad set?)
- Time-sensitive data set diverging? (try detrending data)
- Not properly applying feature transforms from training to validation set?
- Bug?

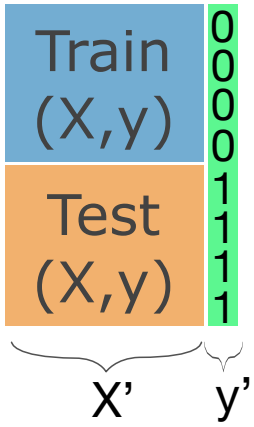
When OOB score is better than validation

- Maybe the validation set is drawn from a different distribution than the training set
- Or, the model is overfit to the data in the training set, focusing on relationships that are not relevant to the test set (e.g., dropping SalesID transaction ID from training set improved our RF model as SalesID never seen in valid set but predictive in training set)
- (Sometimes the validation score is a bit better or worse than the OOB score, due to random fluctuations caused by the inherent randomness of RF construction)

Comparing training / validation sets

Awesome but not well-known trick

- Process to see if valid (or test) set is distinguishable from train set
  1. Combine both X and y from train & valid sets into single data set
  2. Create new target column called **istest** (y' in illustration)
  3. Train model on the combined set (X', y')
  4. Assess metric
- If train/valid are drawn from same distribution, should get 0 metric since there's no way to distinguish between them
- But if you get a high metric, say, 0.95, that applies that train/valid sets are drawn from very different distributions
- TODO: do we compute train score or extract validation set? Maybe use col of random {0,1} as null distribution then see if train score on real 000111 is above that.



If train/test are easily distinguishable…

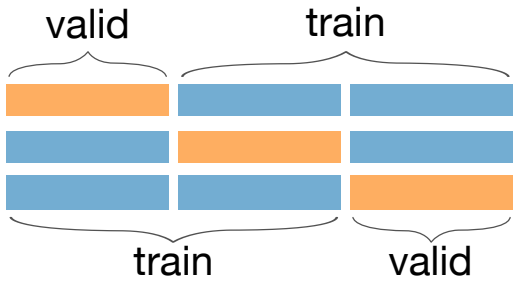
- You might find that ID or date var is different in train vs valid set
- Drop that feature and see how the validation score changes
- Or if y steadily climbs, and all y's are bigger in the validation set, a simple inequality distinguishes training and test set
- Look at the feature importances of original and **istest** models. Features that are important in both are the problem
  - If it's not important in original model, we don't care about it: it's not predictive of target
  - If it's not important in istest model, it's not causing confusion between the data sets
  - Imagine vehicle age is predictive in training and to distinguish train/test; what if age=0 for all in test set? It's difference between used (train) and new cars (test)

Leakage: What if your model is too good?

- Be suspicious if you get an excellent metric; yes, you are awesome, but near-perfect scores are often sign of bug or leakage
- Kaggle example: Uni Melbourne (predict grants)
  - Jeremy Howard (who won) told me that one of the data fields leaked information about whether grant was awarded or not
  - Explanatory variable was only filled in if grant was awarded
  - A missing value almost perfectly predicted “no grant award”
  - I built basic model and got validation accuracy 87% (red flag)
- Rent price example: price/bedroom is great feature and I got very high R^2; oops, I just incorporated (leaked) target y price into X

https://www.kaggle.com/c/unimelb

Stability of metric values



- Getting a single validation metric is usually not enough because scores can vary from run to run because of outliers and anomalies (even with k-fold)
- Also good to collect and compare different metrics
- Consider score fluctuations in NYC rent data (before cleaning)

	OOB	R^2	MAE	MSE
0	0.582	0.002	1,150.354	2,402,630,918.659
1	0.027	0.050	878.512	2,053,053,487.605
2	-0.309	0.323	408.299	3,852,177.529
3	-0.152	-44.812	527.185	265,146,585.910
4	-0.155	-0.105	404.700	7,275,725.459

Outliers cause mismatch between train/valid sets; k-fold, random subsets will see high variability

See https://github.com/parrt/msds621/blob/master/notebooks/assessment/metrics.ipynb

Regressor metrics

Common regressor metrics

- Mean squared error
  Range 0..∞, units(y)^2, symmetric

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

- Mean absolute value
  Range 0..∞, units(y), symmetric

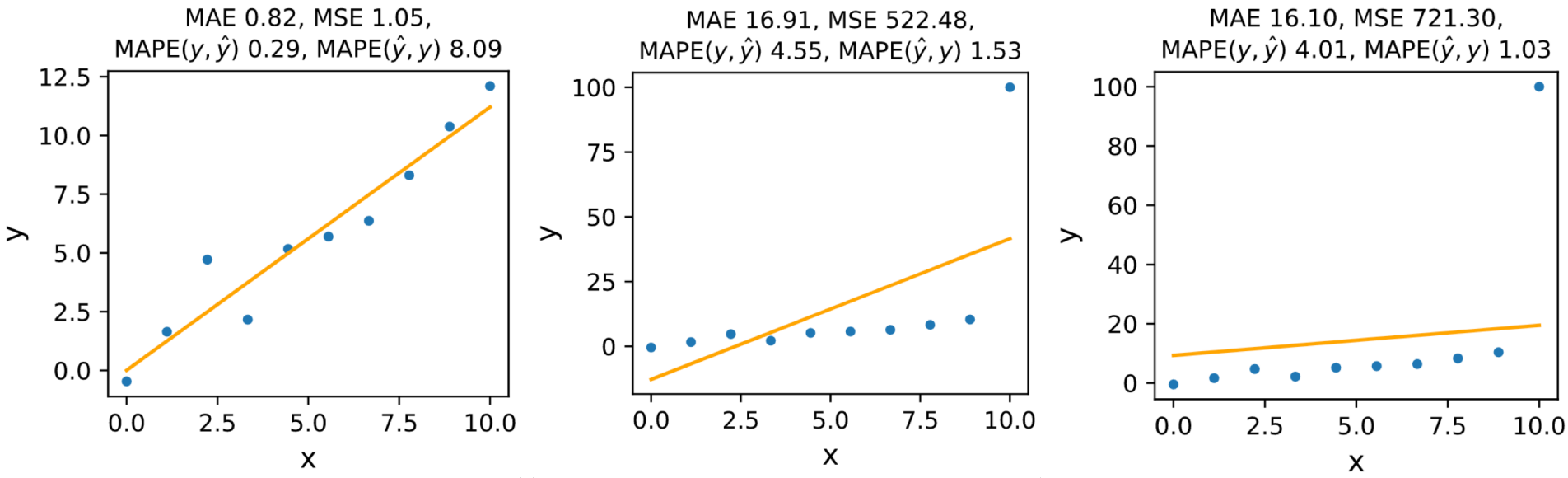
$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- Mean absolute percentage error
  Range 0..∞, unitless, **asymmetric**
  undefined if y=0

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

(For moment, think of symmetry as metric(y, $\hat{y}$)=metric($\hat{y}$, y))

MAE, MSE, MAPE example



MSE incorrectly makes last model look horrible and worse than 2nd model due to outlier.
MAE isn't perfect either as it thinks 2nd and 3rd models are about the same.
Note MAPE asymmetry!

R^2

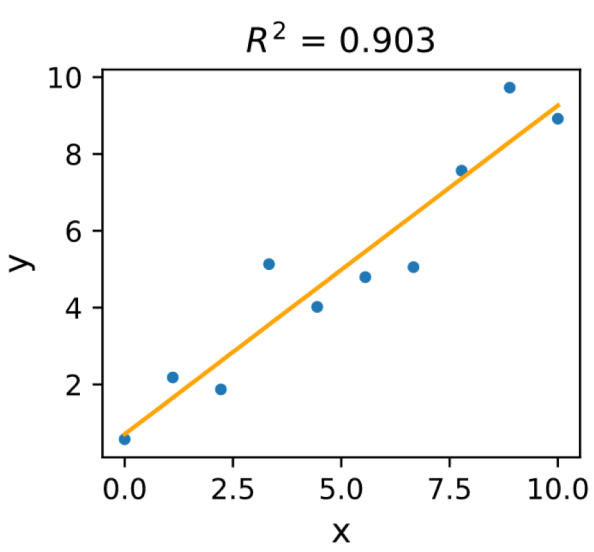
- How well our model performs compared to a “mean model”

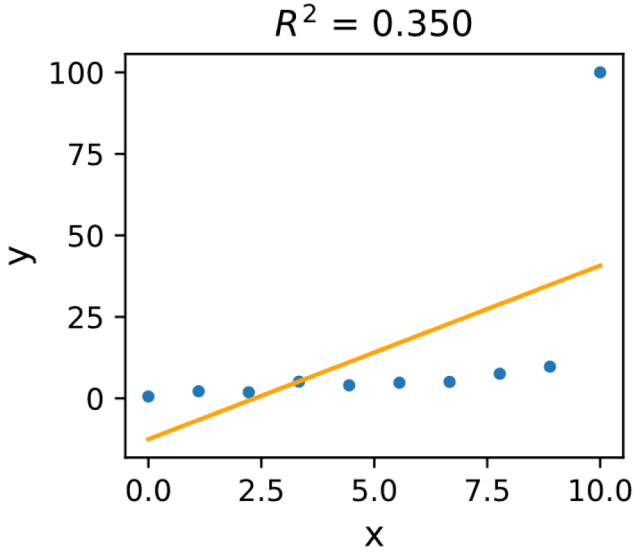
$$R^2 = 1 - \frac{\text{Squared error}}{\text{Variation from mean}} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$$

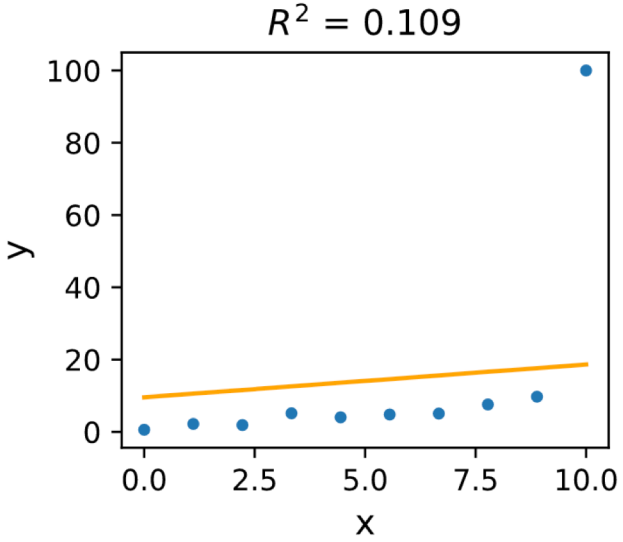
- Range of possible values: $(-\infty, 1]$
- Our model could be really bad, giving large negative numbers
- For OLS linear models, R^2 in [0,1]
- R^2 is default regressor metric for sklearn

Low R^2 doesn't always imply bad model

- Not best metric; here are 3 graphs with good, bad, decent fits
- The 3rd has lowest R^2 but it's a way better model than 2nd

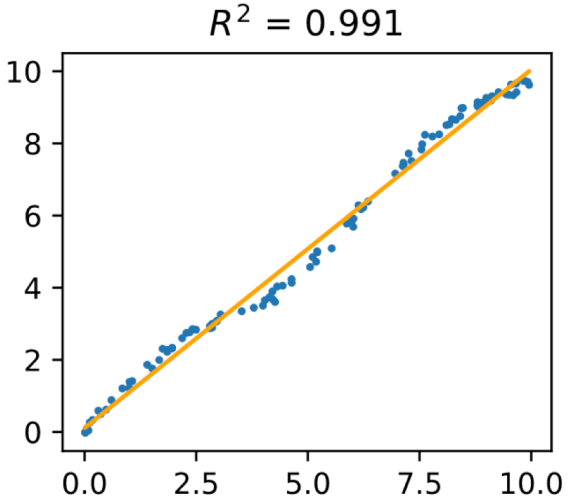


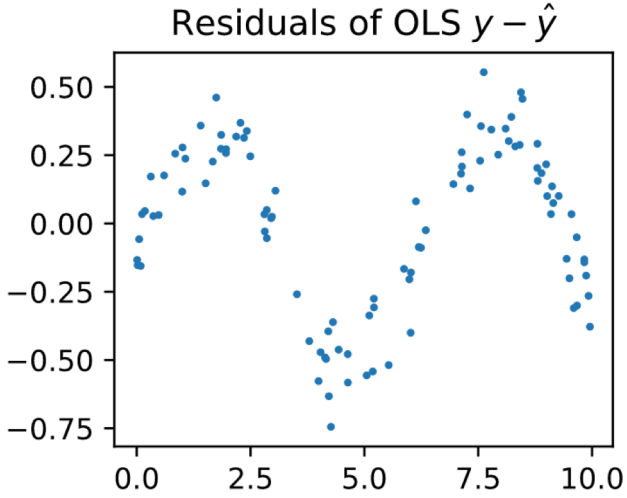




High R^2 doesn't always imply good model

- This linear model looks pretty good and has R^2 = 0.991
- But check the residual plot! Model doesn't capture nature of x,y





See also https://statisticsbyjim.com/regression/interpret-r-squared-regression/

Which metric should we use?

- That depends what we care about for business reasons
- For prices, we usually care more about the percentage error than the absolute amount
- $500 off for a $1,000 apartment is 0.5x or 1.5x off and a big deal but $500 off for a $1 million apartment is a trivial difference
- For things like body temperature that will most likely all be within a small range, the mean absolute error (MAE) is a good and interpretable measure
- The percentage error can be interpretable but there are problems with it: asymmetry
- R^2, MSE and in general squaring error is super sensitive to big deltas

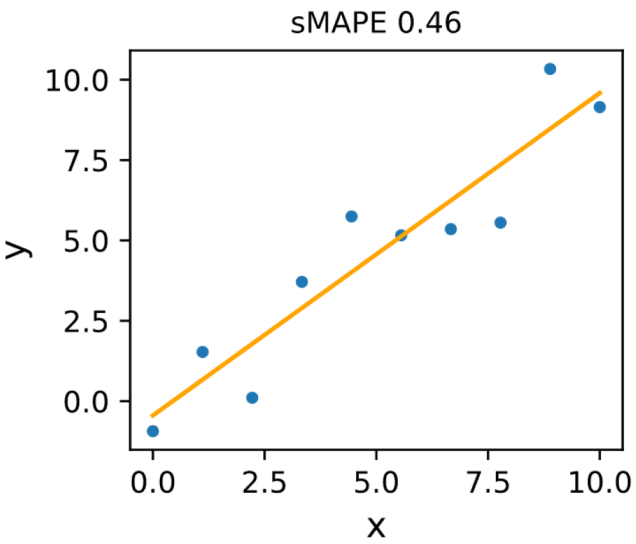
Symmetry in metrics

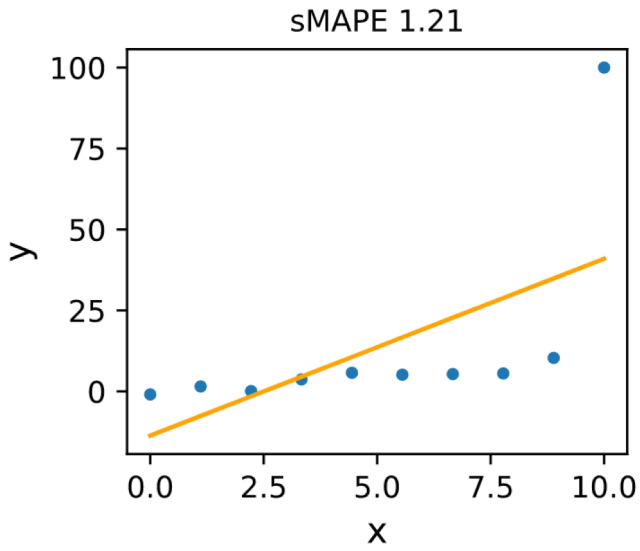
- Most metrics use $y - \hat{y}$ but ratio $y/\hat{y}$ is often better
- But, most ratio-based metrics are asymmetric, which is bad!
  - If y=100, $\hat{y}$=0.01: MAPE = 0.9999
  - If y=0.01, $\hat{y}$=100: MAPE = 9999
- Try symmetric MAPE
  Range 0..2, unitless, symmetric, undefined at y=0 and $\hat{y}$ =0
  (have to work around those landmines)
  - If y=100, $\hat{y}$=0.01: sMAPE = 1.9996
  - If y=0.01, $\hat{y}$=100: sMAPE = 1.9996

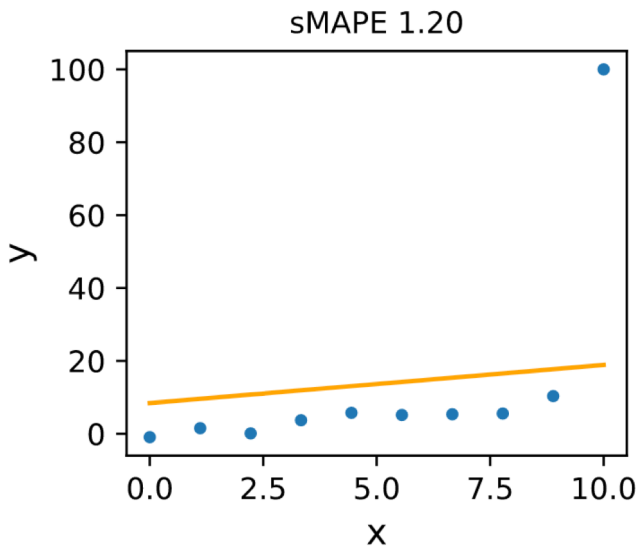
$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

$$sMAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{\frac{1}{2}(|y_i| + |\hat{y}_i|)}$$

sMAPE in action







Classifier metrics

Common classifier metrics

(Ugh; much more complicated than for regressors)

- *TP* = true positive, *TN* = true negative
- *FP* = false positive, *FN* = false negative
- *Confusion matrix* for binary classification to right, but can have larger confusion matrices in general
- The matrices are clear but don't give single metric
- *Accuracy* = correct classification rate = (TN+TP)/n
- *Misclassification* rate is 1 – accuracy

		Predicted F	*Predicted* T
Actual	**F**	TN	FP
	T	FN	TP

		Predicted F	*Predicted* T
Actual	**F**	35	3
	T	1	75

(breast cancer RF)

See also https://en.wikipedia.org/wiki/Evaluation_of_binary_classifiers

True/False positive/negative **rates**

		Predicted F	T
Actual	**F**	TN	FP
	T	FN	TP

- *True positive rate* is TP / num-positive (also called *recall*)
  Of the actually positive y, how many true positives in $\hat{y}$?
  - **TPR** = TP / true-y = TP / (TP + FN)
  - TP over sum of 2nd row in confusion matrix (num true-y is constant)
- *False positive rate* is FP / num-negative
  Of the actually false y, how many false positives in $\hat{y}$?
  - **FPR** = FP / false-y = FP / (FP + TN)
  - FP over sum of 1st row in confusion matrix (num false-y is constant)

Multi-class confusion matrix

- For C classes, we get C x C matrix
- Optimally, it's a diagonal matrix (correct classifications)
- Example; interest in NYC apartments (lo/med/hi); matrix indicates model is good at predicting low interest apts but not others
- Should focus on features that are predictive of med/high to improve model

	predicted_low	predicted_medium	predicted_high
expected_low	3749	566	83
expected_medium	854	418	119
expected_high	189	174	141

More classifier metrics

(I like Precision/recall)

- Precision/recall useful in binary classification, such as spam
- *Precision* = TP/(TP+FP) “of those predicted as positive, how many did we get right?”
- *Recall* = TP/(TP+FN) “of the positive samples, how many did we find (predict as positive)?”
- *F1* is harmonic mean of precision and recall; F1 = 2*(P*R)/(P+R), which gives equal importance to precision and recall

		Predicted F	*Predicted* T
Actual	F	35	3
	T	1	75

precision= 75/(75+3)

		Predicted F	*Predicted* T
Actual	F	35	3
	T	1	75

recall= 75/(75+1)

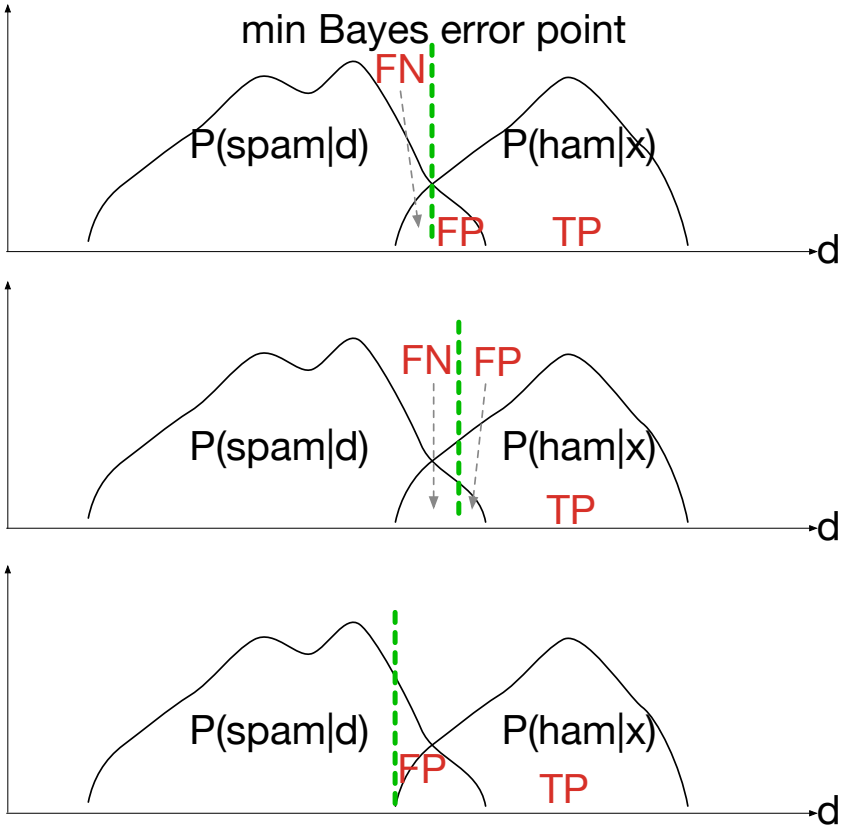
See also https://en.wikipedia.org/wiki/Precision_and_recall

AUC ROC

(area under curve, receiver operator curve)

- Measures separability of distributions
- Encapsulates tradeoff in binary classification between **true positive rate** (recall) and **false positive rate** according to score or likelihood threshold
- Recall logistic regression provides probability of class=1 (ham) and we must choose threshold, which then assigns predicted classes
- Shift threshold left/right and you shift (TP, FP) and (TPR, FPR) coordinates

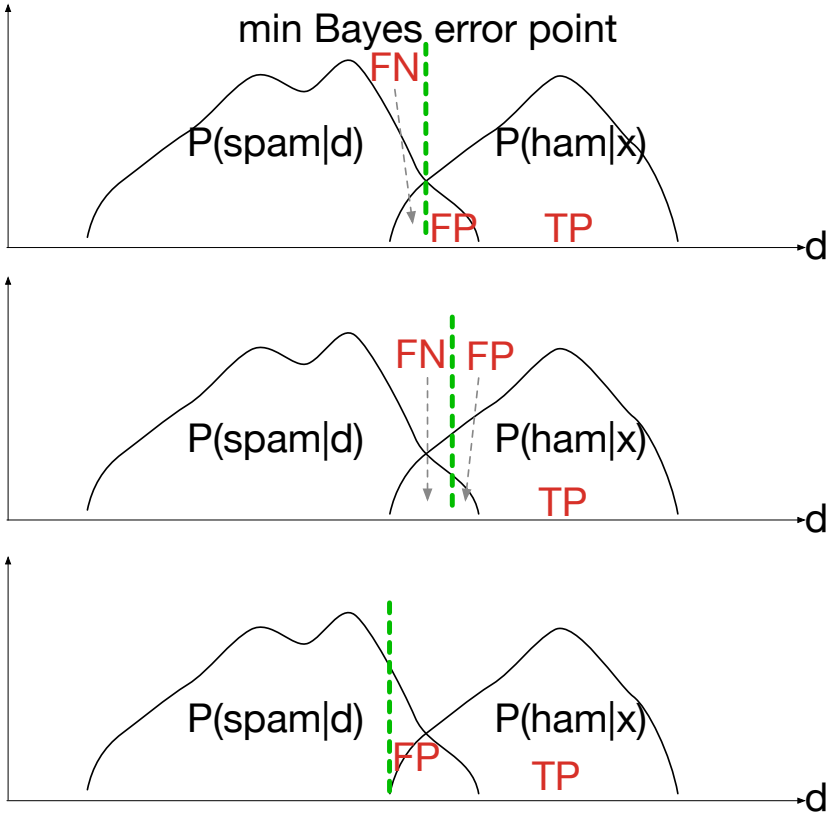
(spam is negative, ham is positive)



See also https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

AUC ROC Continued

- For bayes (min error) threshold, we get one (FPR, TPR) coordinate
- Shift threshold to right, FPR, TPR get smaller and we get another coor.
- Shift to left so FN=0, then:
  TPR = TP/TP = 1.0, but FPR goes up too

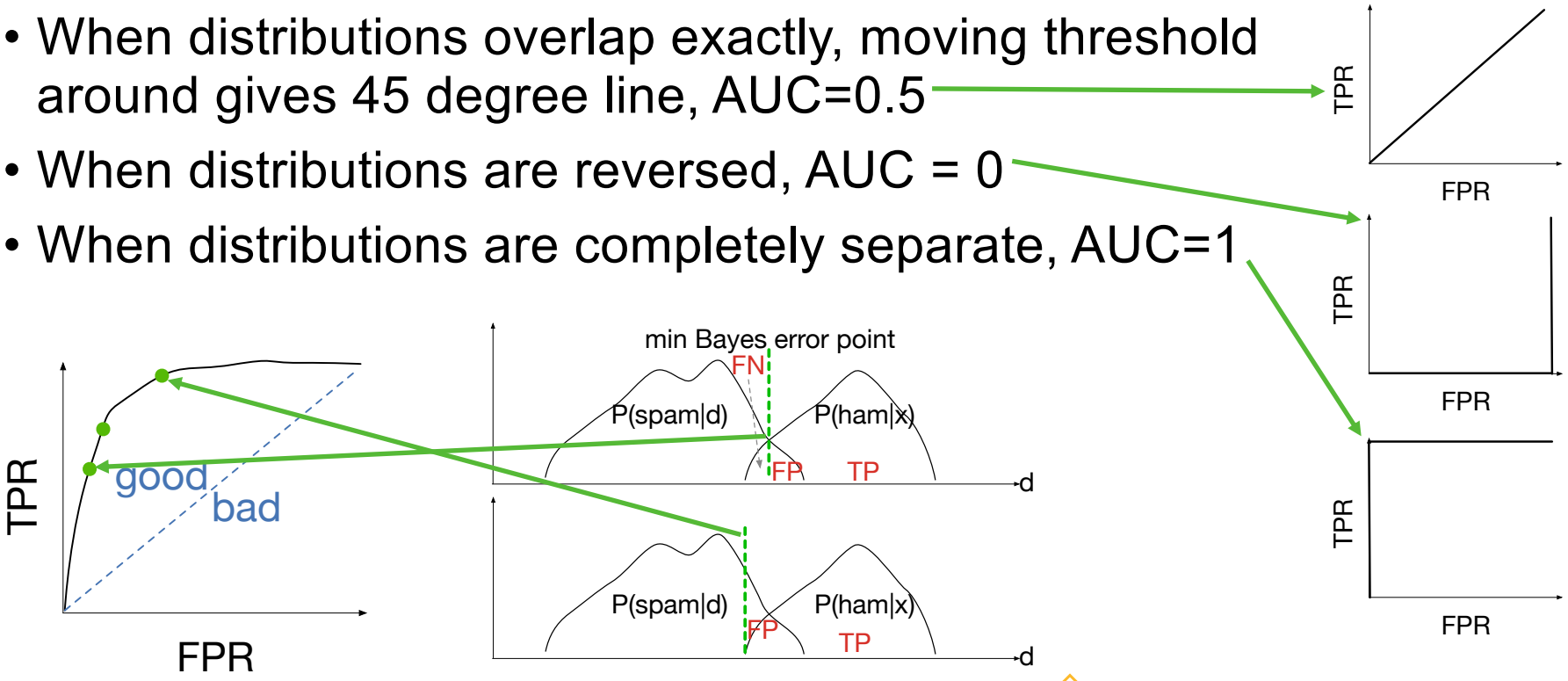


		Predicted F	T
Actual	**F**	TN	FP
	T	FN	TP

TPR = TP / (TP + FN)
FPR = FP / (FP + TN)

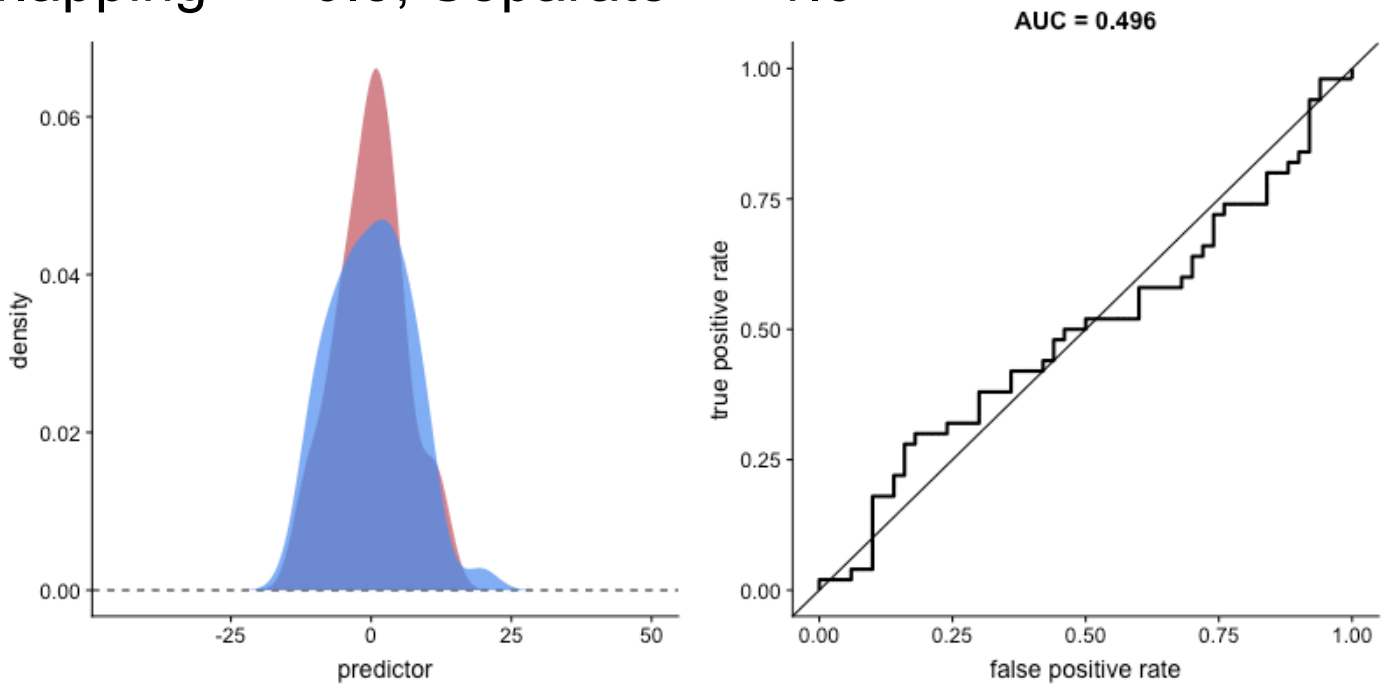
AUC ROC Continued

- When distributions overlap exactly, moving threshold around gives 45 degree line, AUC=0.5
- When distributions are reversed, AUC = 0
- When distributions are completely separate, AUC=1



AUC is a measure of class separability

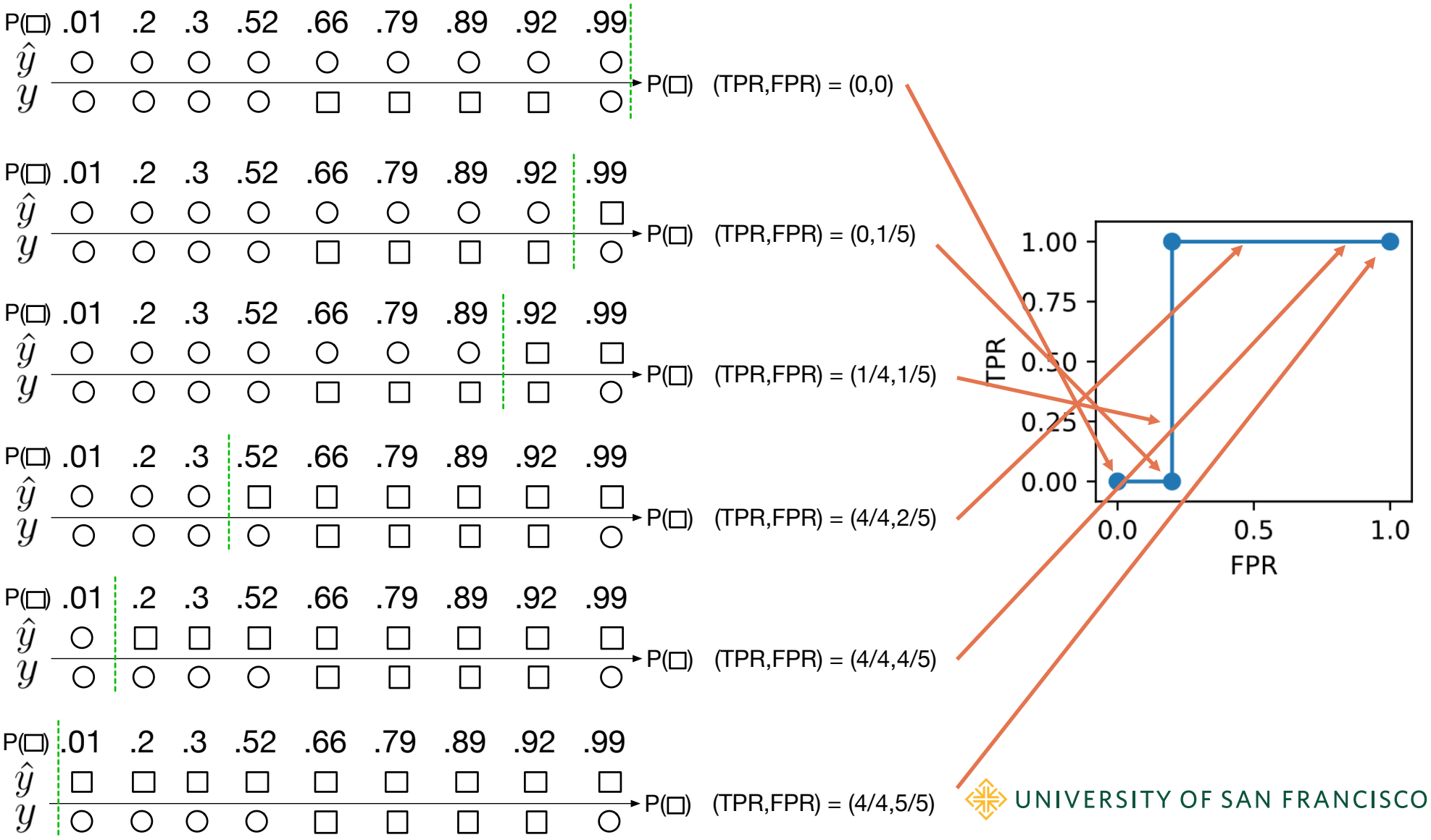
- Overlapping => 0.5, Separate => 1.0



Animation credits: https://github.com/dariyasydykova/open_projects/blob/master/ROC_animation/README.md

Computing ROC (or other curves)

- Requires true y and P(y=1|x), which comes from predict_proba()
- Sort by probabilities then slide a threshold from top to bottom
- Compute confusion matrix and TPR,FPR at each point, which forms ROC curve
- Example next page


P(□) .01 .2 .3 .52 .66 .79 .89 .92 .99
ŷ
y
P(□)
(TPR,FPR) = (0,0)
(TPR,FPR) = (0,1/5)
(TPR,FPR) = (1/4,1/5)
(TPR,FPR) = (4/4,2/5)
(TPR,FPR) = (4/4,4/5)
(TPR,FPR) = (4/4,5/5)
TPR
FPR
1.00
0.75
0.50
0.25
0.00
0.0
0.5
1.0
UNIVERSITY OF SAN FRANCISCO

Code to plot ROC from previous slide

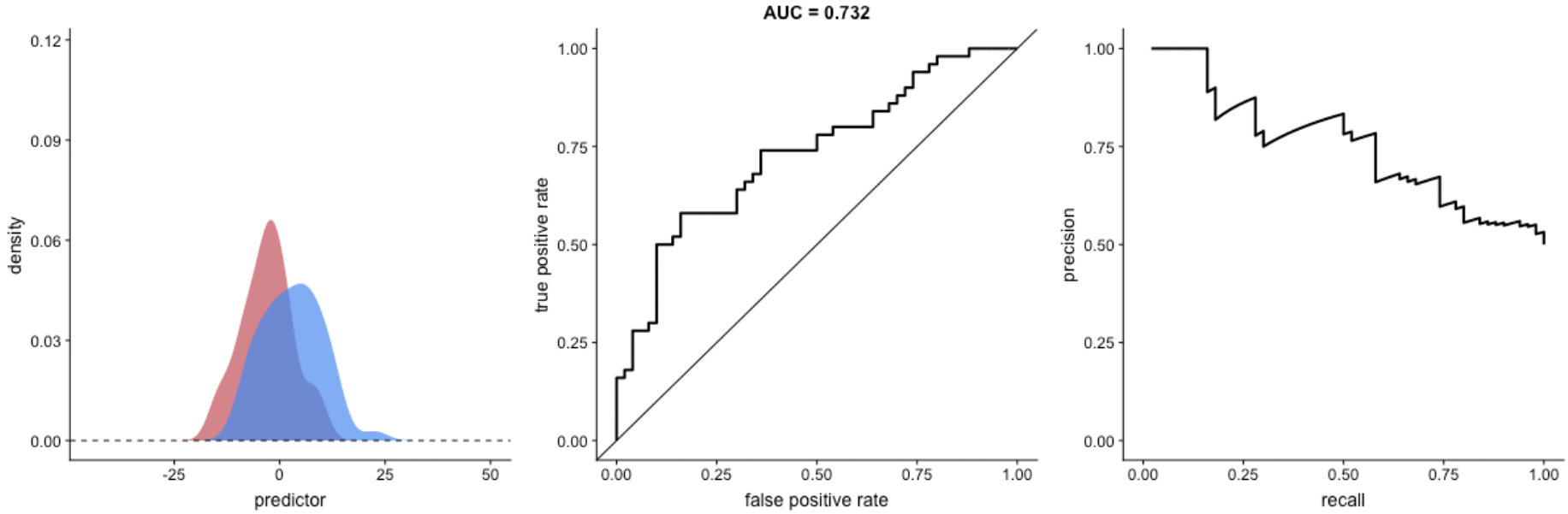
```
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'svg'

y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 0])
yprob = np.array([0.01,.2,.3,.52,.66,.70,.75,.76,.80])
fpr, tpr, thresholds = metrics.roc_curve(y, yprob, pos_label=1)

plt.figure(figsize=(2.2,2))
plt.plot(fpr,tpr)
plt.scatter(fpr,tpr)
plt.xlabel("FPR")
plt.ylabel("TPR")
```

ROC vs Precision-Recall curve

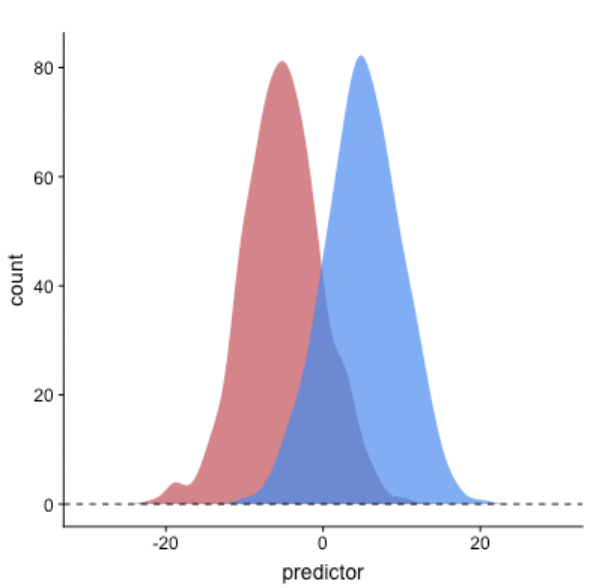
- As variance tightens, ROC goes ups whereas PR goes down

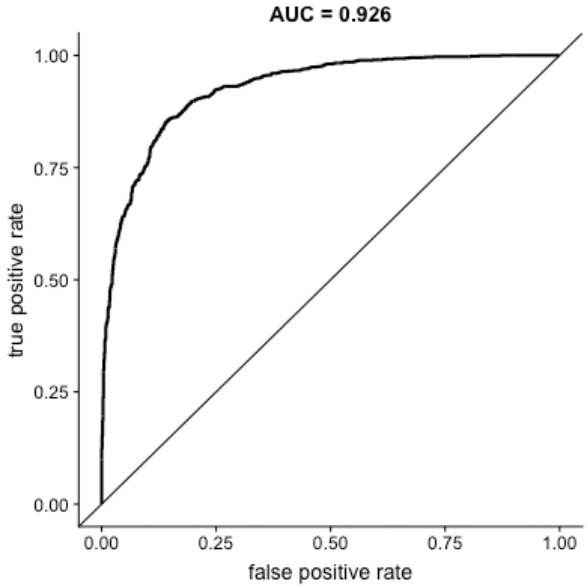


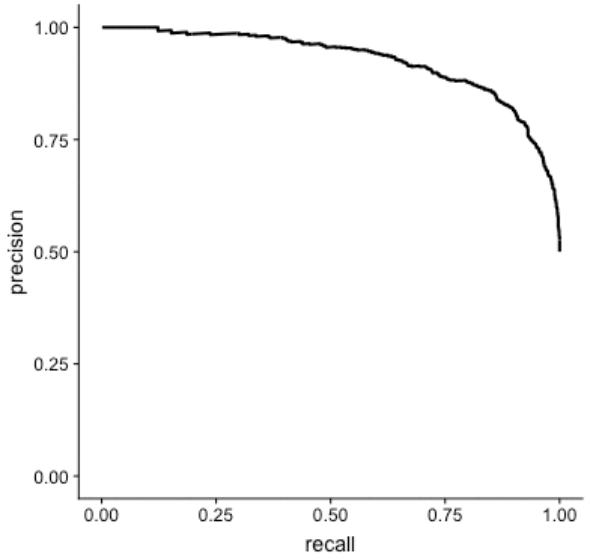
Animation credits: https://github.com/dariyasydykova/open_projects/blob/master/ROC_animation/README.md

Imbalanced classes ROC vs PR curve

- Spam, fraud detection problems are imbalanced; can't trust ROC!
- For same threshold/mean of distribution, only PR curve changes!
- In the end, I favor PR not ROC







Animation credits: https://github.com/dariyasydykova/open_projects/blob/master/ROC_animation/README.md

Upsampling minority class(es) in imbalanced data sets

Credit card data: https://www.kaggle.com/mlg-ulb/creditcardfraud

Naïve use of train_test_split() @ 20%

Imbalanced dataset: Kaggle credit card fraud

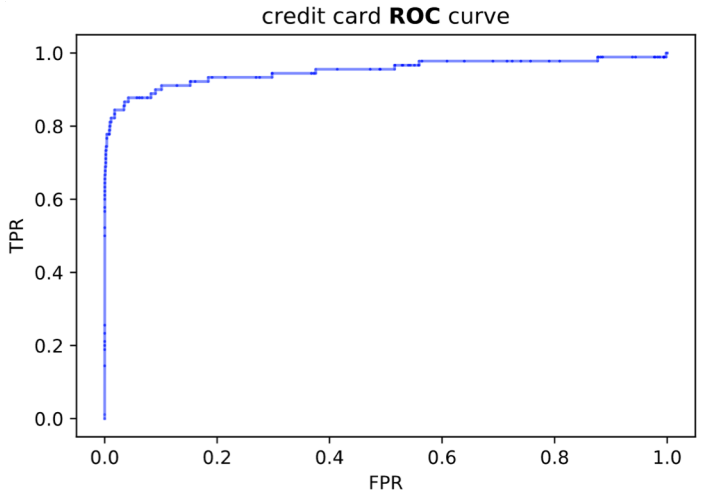
- Num anomalies 492/284807 = 0.17%
- L2 Regularized **LogisticRegression** model

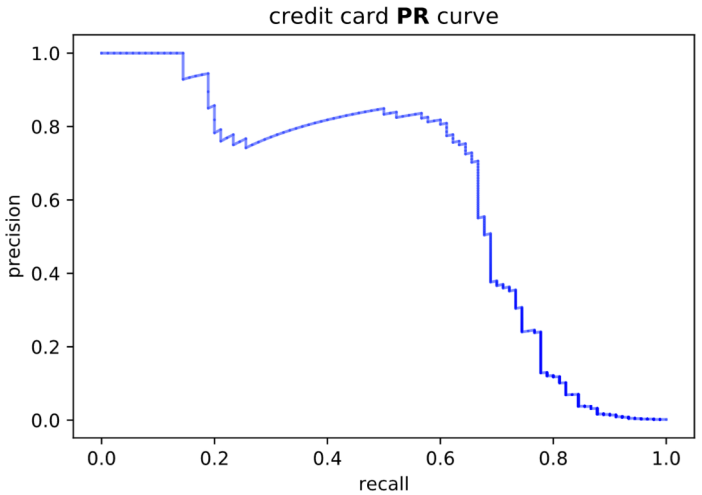
bad
- Accuracy 1.00, AUC ROC 0.93

good
- F1 0.69, AUC PR 0.62

actual \ predicted	F	T
F	56840	16
T	42	64

20% test set





Why is accuracy=1.0?
(That's rounded up)

See https://github.com/parrt/msds621/blob/master/notebooks/assessment/imbalanced.ipynb

First: Proper split to ensure 20% minority class

- Must split out test set before modeling but get 20% from each class at original ratio of class 1 / class 0 (particularly important for small n)
- If num anomalies 492/284807 = 0.17%, then need:
  - TRAIN num fraud 393/227845 = 0.17%
  - TEST num fraud 99/56962 = 0.17%

- Accuracy 1.00, AUC ROC 0.93
- F1 0.70, AUC PR 0.65

	F	T
F	56848	15
T	37	62

```
df_good = df[df['Class']==0]
df_fraud = df[df['Class']==1]

df_train_good, df_test_good   = train_test_split(df_good, test_size=0.20)
df_train_fraud, df_test_fraud = train_test_split(df_fraud, test_size=0.20)

df_train = pd.concat([df_train_good, df_train_fraud], axis=0)
df_test  = pd.concat([df_test_good, df_test_fraud], axis=0)
```

Then: upsample minority in training set

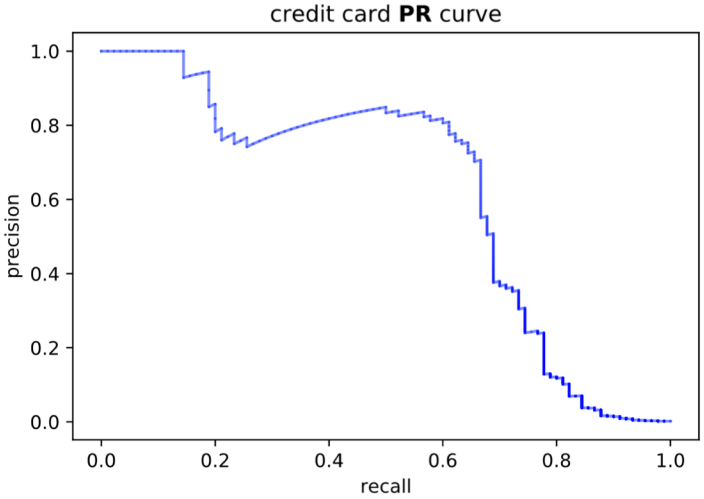
- Upsampled TRAIN num fraud 3930/231382 = 1.70%
- TEST num fraud 99/56962 = 0.17%

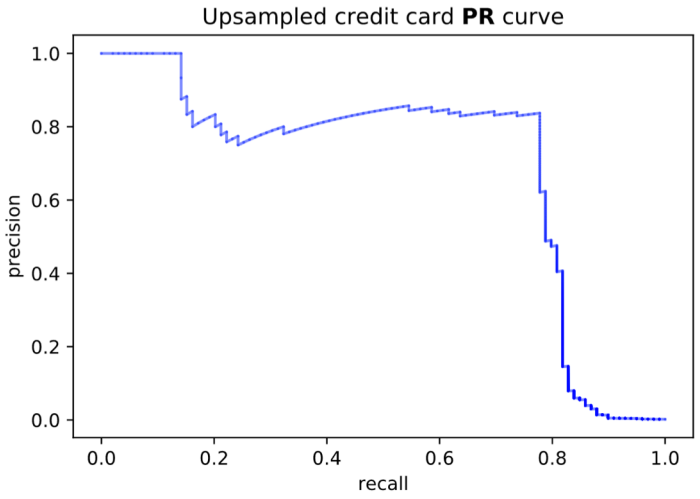
```
df_good = df_train[df_train['Class']==0]
df_fraud = df_train[df_train['Class']==1]

df_fraud_balanced = df_fraud.sample(int(len(df_fraud)*10), replace=True)
df_train_upsampled = pd.concat([df_good, df_fraud_balanced], axis=0)
```

Upsampling fraud 10x in training data

- We get improvement with upsampling for logistic regression, at least for this Kaggle credit card data set





- Accuracy 1.00, AUC ROC 0.95
- F1 0.73, AUC PR 0.70

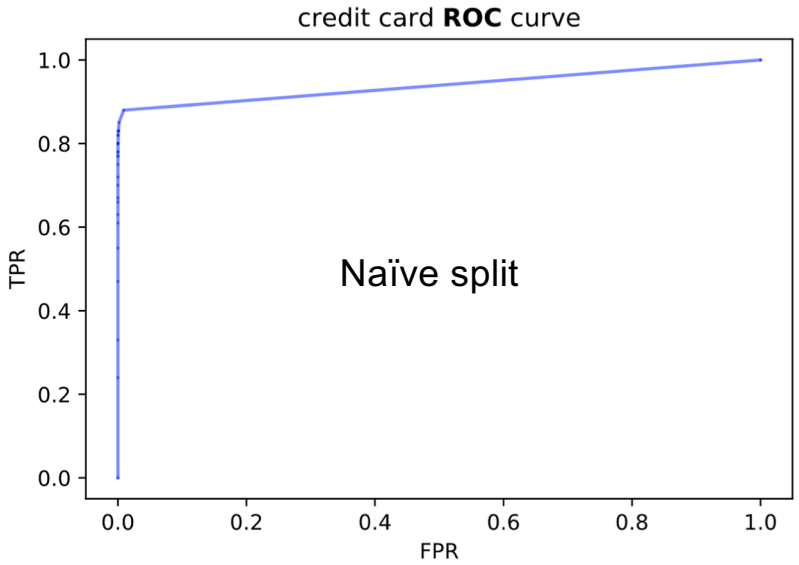
	F	T
F	56827	36
T	22	77

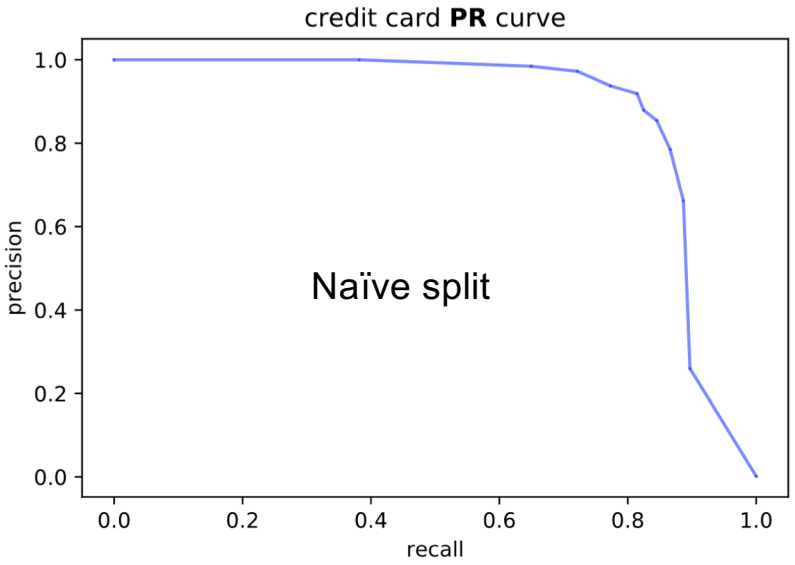
L2 Logistic Regression summary

- Naive split
  - F1 0.69, Accuracy 1.00
  - ROC 0.93, AUC PR 0.62
- Proper split (just making sure train/test balance is right can help)
  - F1 0.70, Accuracy 1.00
  - AUC ROC 0.93, AUC PR 0.65
- Upsample:
  - F1 0.73, Accuracy 1.00
  - AUC ROC 0.95, AUC PR 0.70
- (Varies depending on actual test set held out)

Credit card fraud with RF (10 trees)

- RFs do better than logistic: RF better at isolating minority samples
  - Naïve: ROC 0.92, AUC PR 0.84
  - Balanced split: AUC ROC 0.97, AUC PR 0.93
  - 10x upsampled training: AUC ROC 0.97, AUC PR 0.94 (doesn't really help)





	F	T
F	56860	3
T	13	86

Penalizing inappropriate confidence

- Terence's cousin went to the doctor with suspected skin cancer
- Doc was "100% positive mole was benign"
- 6 months later cousin loses most of her tricep / upper arm
- Doc's model was seriously flawed but not necessarily because of diagnosis
- The biggest mistake was the certainty of the incorrect diagnosis, as it allowed the cancer to spread
- Less certainty would've provided opportunities for more tests…
- Same concept of model assessment applies to ML models

Log loss (is both metric and loss function)

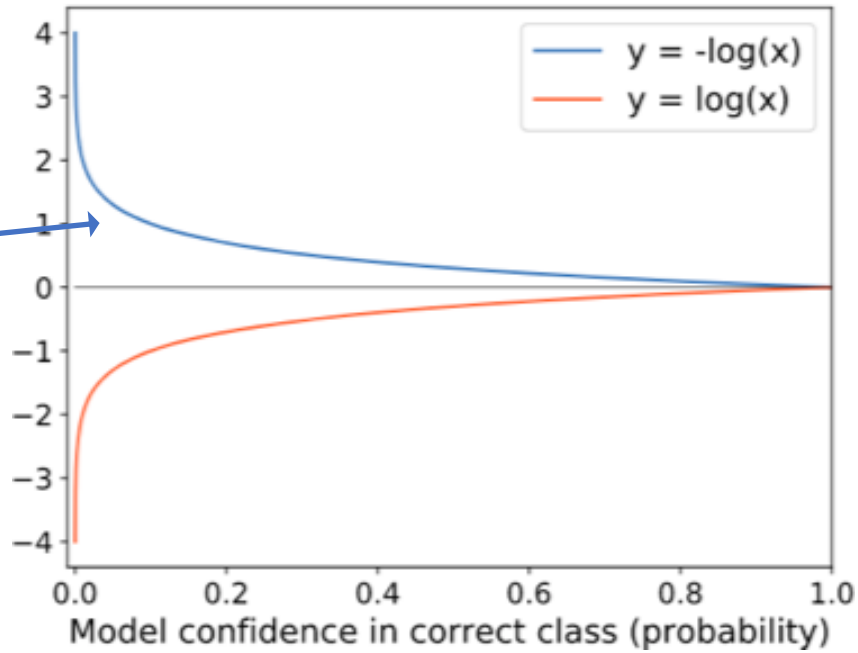
- Measures model performance when model gives probabilities, like logistic regression but RFs can give probabilities too
- Works for any number of classes; we'll do binary only
- Penalizes very confident misclassifications strongly
- Perfect score is 0 log loss, imperfection gives unbounded scores
- Let p be probability of true class, such as fraud or cancer; 1-p is then probability of false class, such as not fraud, not cancer
- Log loss is function of actual y and estimated p, not predicted class

Log loss continued

- Assume model gives us p (prob cancer) predicted from some x
- Case 1: true y is cancer
  - If p=0.9, we are confident it's cancer and rightly so: loss should be low
  - If p=0.01, we are confident it's NOT cancer and wrongly so: **penalize** with high (i.e., bad) loss value
- Case 2: true y is benign (not cancer)
  - If p=0.9, we are confident it's cancer and wrongly so: **penalize**
  - If p=0.01, we are confident it's NOT cancer and rightly so: loss is low
- Let loss = *penalty*(p) if y=1 else *penalty*(1– p)
  where penalty(p) should be very high at low p (confident FP)

Log loss penalty

- loss = *penalty*(p) if y=1 else *penalty*(1– p)
- Let *penalty*(p) = -log(p)



$$loss = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} -\log(p_i) & y = 1 \\ -\log(1-p_i) & y = 0 \end{cases}$$

$$loss = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(p) + (1-y_i)\log(1-p_i)$$

So log loss is average penalty where penalty is very high for confidence in wrong answer