Still KNN...

Data analysis

Classification evaluation

Tricks on Model Selection, Development, and Training

Modeling Process

	Regression ($y \in \mathbb{R}$)	Classification ($y \in \{0, 1\}$)
1. Choose a model ✓	Linear Regression $\hat{y} = f_\theta(x) = x^T\theta$	Logistic Regression $\hat{P}_\theta(Y = 1\|x) = \sigma(x^T\theta)$
2. Choose a loss function ✓	Squared Loss or Absolute Loss	Average Cross-Entropy Loss $-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log(\sigma(X_i^T\theta)) + (1 - y_i)\log(1 - \sigma(X_i^T\theta))\right)$
3. Fit the model ✓	Sklearn/Gradient descent	Sklearn/Gradient descent
4. Evaluate model performance	R^2, RMSE, Residuals, etc.	Let's see!

Classifier Accuracy

The most basic evaluation metric for a classifier is **accuracy**.

$$\text{accuracy} = \frac{\#\text{ of points classified correctly}}{\#\text{ points total}}$$

```
def accuracy(X, Y):
    return np.mean(model.predict(X) == Y)

accuracy(X, Y) # 0.794
```

```
model.score(X, Y) # 0.794
```

(sklearn documentation)

While widely used, the accuracy metric is **is problematic** when dealing with **class imbalance**.

Pitfalls of Accuracy: Class Imbalance

Suppose we're trying to build a classifier to filter spam emails.

- Each email is **spam** (1) or **ham** (0).

Let's say we have 100 emails, of which only **5** are truly **spam**, and the remaining **95** are **ham**.

Accuracy is not always a good metric for classification, particularly when your data have **class imbalance** (e.g., very few 1's compared to 0's).

Your friend ("Friend 1"):

Classify every email as **ham** (0).

$$\text{accuracy}_1 = \frac{95}{100} = 0.95$$

High accuracy...
...but we detected **none** ⚠ of the spam!!!

Your other friend:

Classify every email as **spam** (1).

$$\text{accuracy}_2 = \frac{5}{100} = 0.05$$

Low ⚠ accuracy...
...but we detected **all** of the spam!!!

Types of Classifications

There are four different classifications that our model might make:

- True positive: correctly classify a positive point as being positive $y = 1, \hat{y} = 1$
- True negative: correctly classify a negative point as being negative $y = 0, \hat{y} = 0$
- False positive: incorrectly classify a negative point as being positive $y = 0, \hat{y} = 1$
- False negative: incorrectly classify a positive point as being negative $y = 1, \hat{y} = 0$

"**positive**" means a prediction of **1**.
"**negative**" means a prediction of **0**.

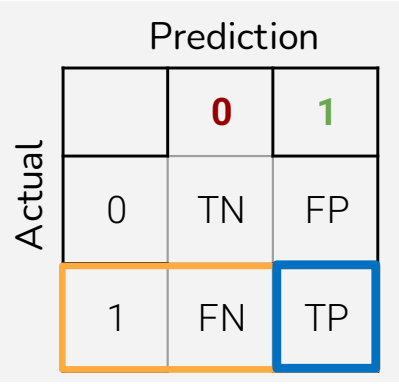
A confusion matrix plots these quantities for a particular classifier (threshold) and dataset.

Actual y \ Prediction $\hat{y}$	**0**	**1**
0	True **negative** (TN)	False **positive** (FP)
1	False **negative** (FN)	True **positive** (TP)

Accuracy, Precision, and Recall

$$\text{accuracy} = \frac{TP + TN}{n}$$

What proportion of points did our classifier classify correctly?



Precision and **recall** are two commonly used metrics that measure performance even in the presence of class imbalance.

$$\text{precision} = \frac{TP}{TP + FP}$$

Of all observations that were predicted to be 1, what proportion were actually 1?

- How accurate is our classifier when it is positive?
- Penalizes false positives.

Same Numerator

$$\text{recall} = \frac{TP}{TP + FN}$$

Of all observations that were actually 1, what proportion did we predict to be 1? (Also known as sensitivity.)

- How **sensitive** is our classifier to **positives**?
- Penalizes false negatives.

Back to the Spam

$$\text{accuracy} = \frac{TP + TN}{n}$$
$$\text{precision} = \frac{TP}{TP + FP}$$
$$\text{recall} = \frac{TP}{TP + FN}$$

Suppose we're trying to build a classifier to filter spam emails.

- Each email is **spam** (1) or **ham** (0).

Let's say we have 100 emails, of which only **5** are truly **spam**, and the remaining **95** are **ham**.

Your friend:

Classify every email as **ham** (0).

$$\text{accuracy}_1 = \frac{95}{100} = 0.95$$

Never positive!

$$\text{precision}_1 = \frac{0}{0+0} = \text{undefined}$$

$$\text{recall}_1 = \frac{0}{0+5} = 0$$

"Don't Recall"

Your other friend ("Friend 2"):

Classify every email as **spam** (1).

	0	1
0	TN: 0	FP: 95
1	FN: 0	TP: 5

$$\text{accuracy}_2 = \frac{5}{100} = 0.05$$

$$\text{precision}_2 = \frac{5}{5+95} = 0.05$$

Many false positives!

$$\text{recall}_2 = \frac{5}{5+0} = 1.0$$

No false negatives!

Precision vs. Recall

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$

Precision penalizes false positives and Recall penalizes false negatives.

There is a **tradeoff** between precision and recall; they are often inversely related.

- Ideally, both would be near 100%, but that's unlikely to happen.

In many settings, there might be a **higher "cost"** to missing **positive** or **negative** cases.

Some examples:

- Detecting if someone tests positive (1) or negative (0) for a disease.
- Determining if someone should be sentenced to prison (1) or not (0).
- Filtering an email as spam (1) or ham (0).

True and False Positive Rates

Keeping things interesting – two more performance metrics.

$$\text{FPR} = \frac{FP}{FP + TN}$$

False Positive Rate (FPR): out of all datapoints that had Y=0, how many did we classify **incorrectly**?

$$\text{TPR} = \frac{TP}{TP + FN}$$

True Positive Rate (TPR): out of all datapoints that had Y=1, how many did we classify correctly? **Same as recall.**

Actual \ Prediction	**0**	**1**
0	TN	FP
1	FN	TP

F1 score

Tries to Balance them... Harmonic mean of the 2

One way to **balance precision and recall** is to maximize the ***F_1* Score:**

$$\text{F1 Score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

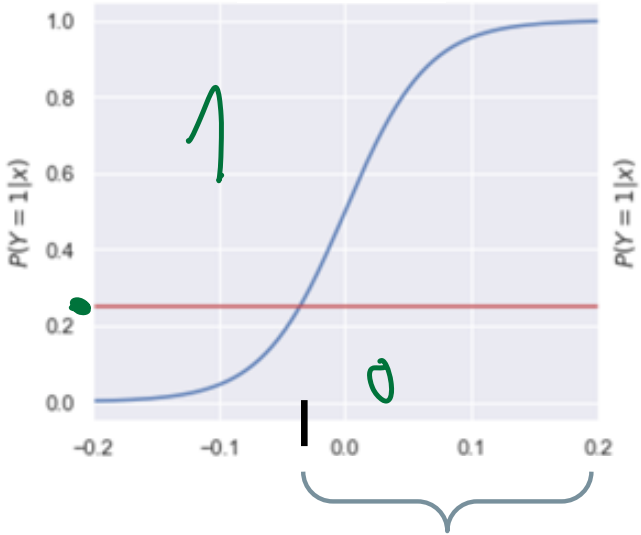
- **The harmonic mean of precision and recall**
- Often used when there is a **large class imbalance**
- Optimizes for the **true-positive** case (does not not optimize **true-negatives)**
  - May not always want to **balance precision and recall**

Changing the Threshold

$$\hat{y} = \text{classify}(x) = \begin{cases} \text{Class 1} & p \geq T \\ \text{Class 0} & p < T \end{cases}$$

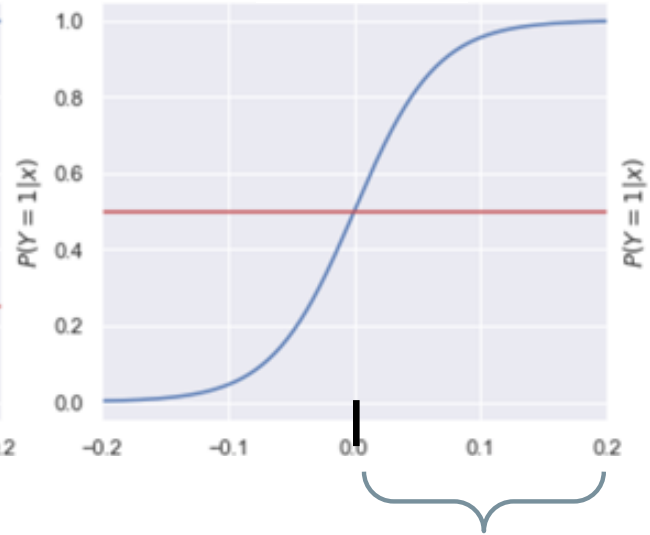
As we increase the threshold T, we “raise the standard” of how confident our classifier needs to be to predict 1 (i.e., “positive”).

T = 0.25



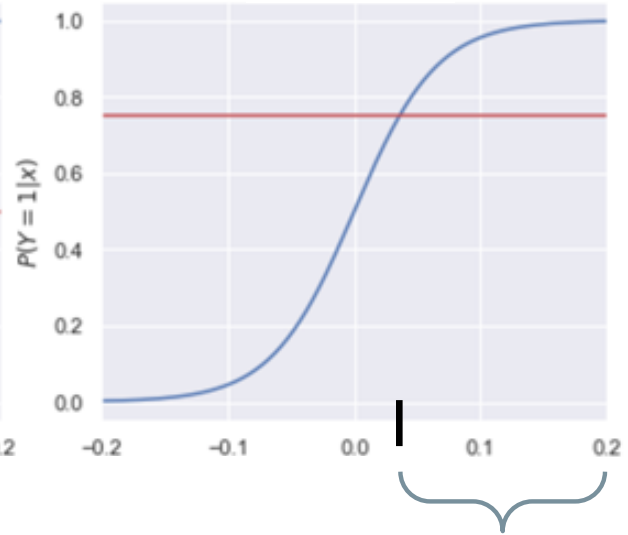
These x will all predict 1

T = 0.50



What we start plotting

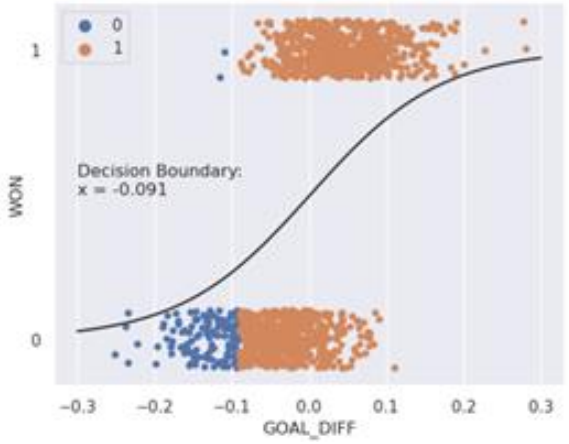
T = 0.75



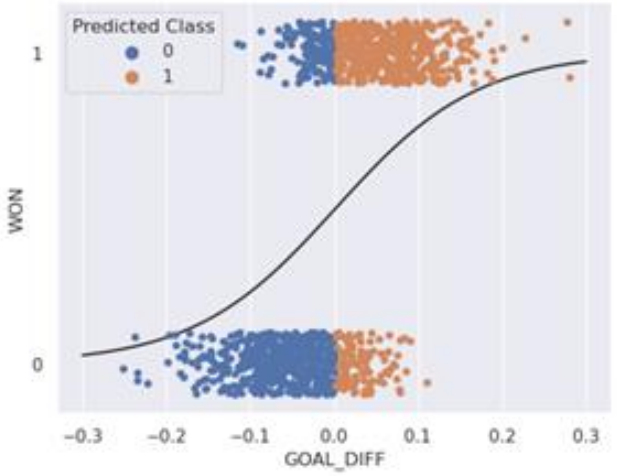
Fewer positives

Changing the Threshold

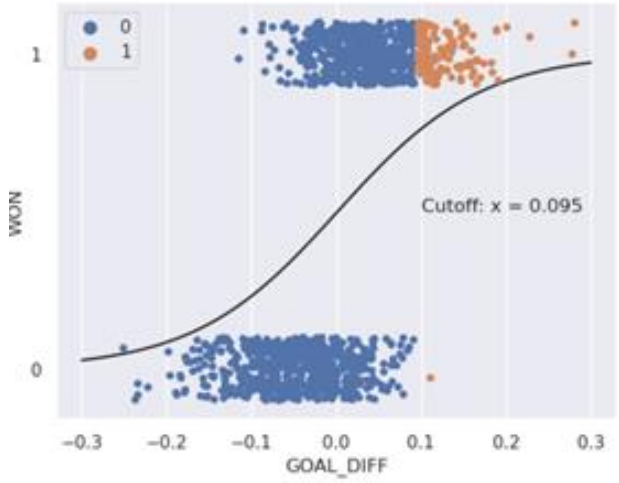
T = 0.25



T = 0.50



T = 0.75



How to interpret a higher classification threshold: the model needs to predict a higher probability of a point belonging to Class 1 before it can confidently classify it as Class 1

- As T increases, we predict fewer positives!

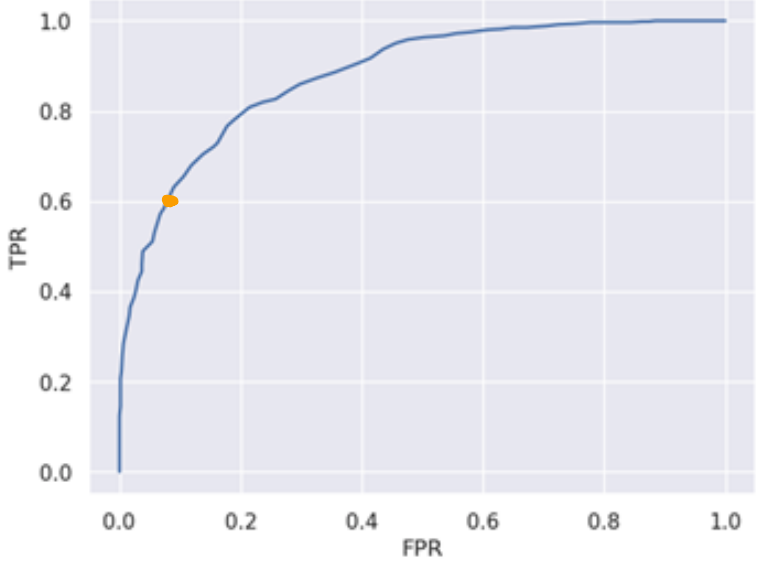
Changing the threshold allows us to finetune how “confident” we want our model to be before making a positive prediction

ROC Curves

We can perform a similar process with FPR and TPR

1) Try many thresholds
2) Compute the FPR and TPR for each threshold
3) Choose a threshold that keeps FPR *low* and TPR *high*

ROC = “receiver operating characteristic”, comes from radar in WWII

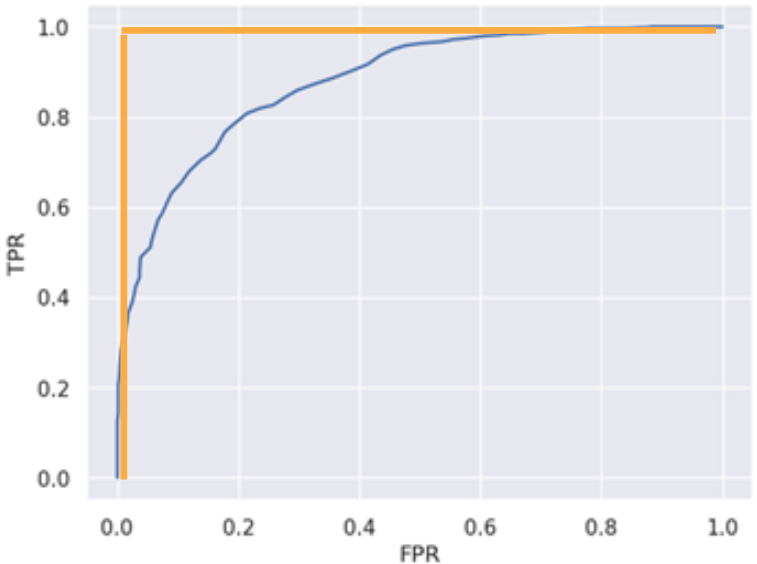


Threshold is low:
FPR high, TPR high
(many positive predictions)

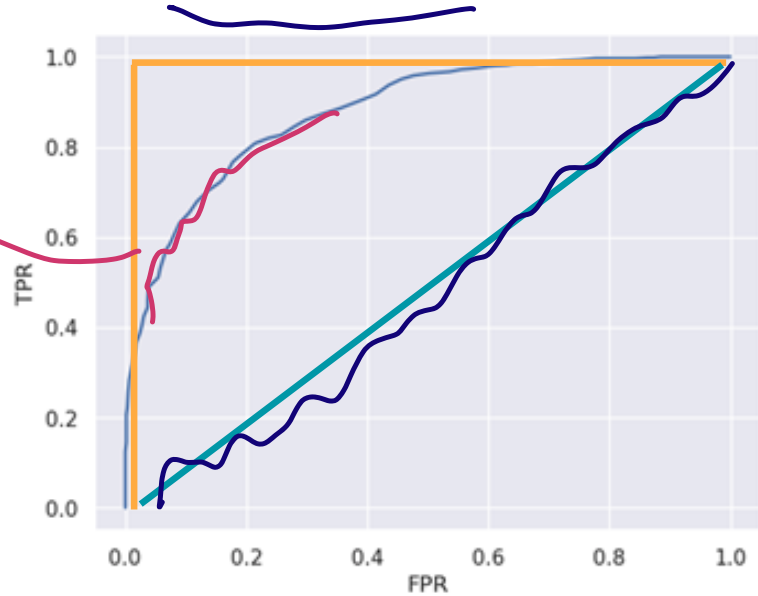
Threshold is high:
FPR low, TPR low
(few positive predictions)

ROC Curves

A perfect predictor has TPR = 1 and FPR = 0



A predictor that **predicts randomly** has an AUC of 0.5



The **Area Under Curve (AUC)** of the perfect predictor is 1.
Because we want our classifier to be as close as possible to the perfect predictor, we aim to maximize the AUC.

Real-world classifiers have an AUC between 0.5 and 1.

Extra topics

1. How to engineer good features
   - Feature importance
   - Feature generalization

2. Model selection

Measuring a feature’s importance

How much the model performance deteriorates
if a feature or a set of features containing that feature
is removed from the model?

Measuring a feature's importance

- Algorithms like XGBoost (coming up) allow us to measure feature importance

get_score(*fmap='', importance_type='weight'*)

Get feature importance of each feature. For tree model Importance type can be defined as:

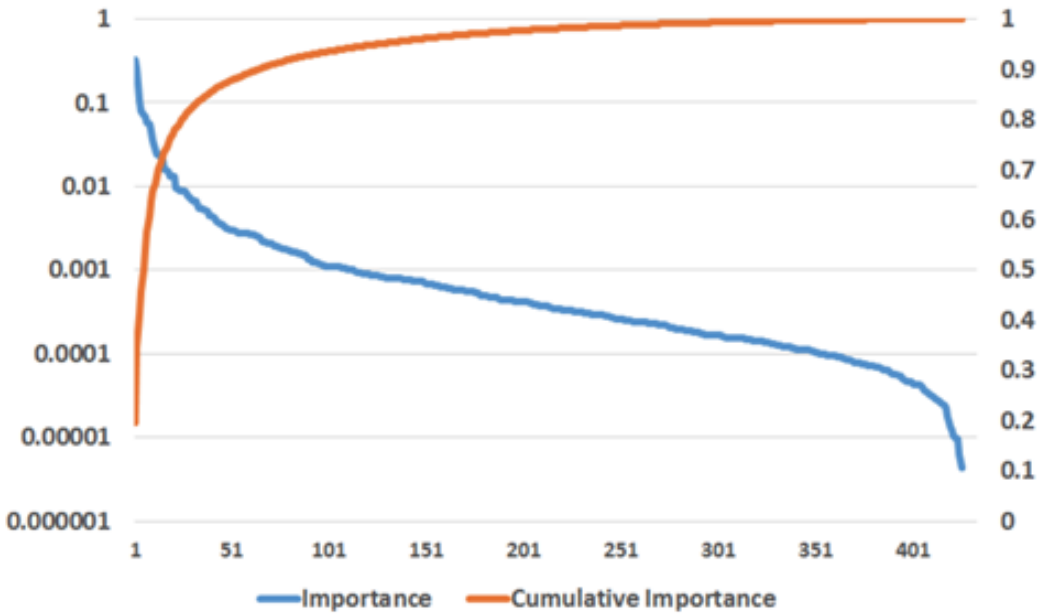
- 'weight': the number of times a feature is used to split the data across all trees.
- 'gain': the average gain across all splits the feature is used in.
- 'cover': the average coverage across all splits the feature is used in.
- 'total_gain': the total gain across all splits the feature is used in.
- 'total_cover': the total coverage across all splits the feature is used in.

The Multiple faces of 'Feature importance' in XGBoost (Amjad Abu-Rmileh, 2019)

https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.Booster.get_score

Measuring feature importance @ Facebook

- Top 10 features: 50% total feature importance
- Bottom 300 features: <1% total feature importance



Practical Lessons from Predicting Clicks on Ads at Facebook (He et al., 2014)

Feature engineering: the more the better?

- Adding more features tends to improve model performance

How can having too many features be bad?

Too many features can be bad ...

- Training:
  - Overfitting
  - More features, more opportunity for data leakage
- Inference
  - Increase inference latency with online prediction
  - Might cause increased memory usage -> more expensive instance required
- Stale features become "technical debts"

Solution:

- Clean up stale / ineffective features
- Store features in case you want to reuse them
  - Feature management

9 best practices for feature engineering

1. Split data by time instead of doing it randomly.
2. If you oversample your data, do it after splitting.
3. Use statistics/info from the train split, instead of the entire data, for feature engineering: scaling, normalizing, handling missing values, etc.
4. Understand feature importance to your model.
5. Measure correlation between features and labels/outputs.
6. Use features that *generalize* well.
7. Remove stale features from your models.

8. Understand how your data is generated, collected and processed.
9. Involve domain experts if necessary.

Model selection

EVERY ML algorithm ever (to be?) built

- Function to be learned
  - E.g. model architecture, number of hidden layers
- Objective function to optimize (minimize)
  - Loss function
- Learning procedure (optimizer)
  - Gradient descent (Adam, Momentum,...)

6 tips for evaluating ML algorithms

1. Avoid the state-of-the-art trap

- SOTA's promise
  - Why use an old solution when a newer one exists?
  - It's exciting to work on shiny things
  - Marketing
- SOTA's reality
  - SOTA on research data != SOTA on your data
  - Cost
  - Latency
  - Proven industry success
  - Community support

2. Start with the simplest models

If it works, it solves your problem

- Easier to deploy
  - Deploying early allows validating pipeline
- Easier to debug
- Easier to improve upon

- Simplest models != models with the least effort
  - Linear regression is always a good starting point

3. Avoid human biases in selecting models

- It’s important to evaluate models under comparable conditions
  - It’s tempting to run more experiments for X because you’re more excited about X

- Near-impossible to make blanketed claims that X is *always* better than Y

4. Better now vs. better later

- Best model now != best model in 2 months
  - Improvement potential with more data
  - Ease of update

5. Evaluate trade-offs

- False positives vs. false negatives
- Accuracy vs. compute/latency
- Accuracy vs. interpretability

6. Understand your model's assumption

- IID
  - **Most of statistical models** assume that examples are **independent and identically distributed**
- Smoothness
  - **Supervised algorithms** assume that there's a set of functions that can transform inputs into outputs such that **similar inputs are transformed into similar outputs**
- Tractability
  - Let X be the input and Z be the latent representation of X. **Generative models** assume that it's tractable to compute P(Z|X).
- Boundaries
  - **Linear classifiers** assume that **decision boundaries are linear**.
- Conditional independence
  - **Naive Bayes classifiers** assume that the **attribute values are independent of each other given the class.**