Day 1

Course Overview & Fundamentals for Machine Learning in Python

HPSI
Human Performance Systems, Inc.

Plan for the Week

- Day 1: Intro & Review
  - Introduction & Course Overview
  - Review of Fundamentals for Machine Learning in Python
- Day 2: Intro to Classification Algorithms
  - Probability Theory
  - Common & Probabilistic Classifiers
- Day 3: Tree-Based Methods for Classification
  - Decision Tree
  - Random Forest
- Day 4: Neural Networks
  - Perceptron (Single & Multilayer) & ADALINE Networks
  - Optimization, Backpropagation, & Learning Rules
- Day 5: Deep Learning (Neural Networks Cont.)
  - Brief Background
  - Convolution Networks
  - Tensorflow & Keras

Learning Objectives

- By the end of the course, students should be able to:
  1. Understand the mathematical and statistical foundations of common classification algorithms
  2. Identify the most appropriate algorithm based on data characteristics and analysis goals
  3. Implement classification algorithms in Python
  4. Evaluate model performance based on classification metrics

Course Structure & Delivery

- Class is from 9:00 AM – 3:30 PM (6.5 hours)
  - This will be broken up into 3 sessions

- Session 1 (2.25 hours)
  - 9:00 AM – 11:15 AM
- Lunch (1 hour)
  - 11:15 AM – 12:15 PM
- Session 2 (2.25 hours)
  - 12:15 PM – 2:30 PM
- Break (10 minutes)
  - 2:30 PM – 2:40 PM
- Session 3 (50 minutes)
  - 2:40 PM – 3:30 PM

HPSI
Human Performance Systems, Inc.

Course Structure & Delivery

- Session 1 (2.25 hours)
  - Introduce Topic → ~ 1 hour
  - Guided Example & Walkthrough → ~ 30 minutes
  - In-class Exercise → ~ 45 minutes
- Session 2 (2.25 hours)
  - Same format
- Session 3 (50 minutes)
  - Introduce Light Topic/Next Day
  - Learning Evaluation
  - Questions/Recap of the Day

Plan for the Week

- Day 1: Intro & Review
  - Introduction & Course Overview
  - Review of Fundamentals for Machine Learning in Python
- Day 2: Intro to Classification Algorithms
  - Probability Theory
  - Common & Probabilistic Classifiers
- Day 3: Tree-Based Methods for Classification
  - Decision Tree
  - Random Forest
- Day 4: Neural Networks
  - Perceptron (Single & Multilayer) & ADALINE Networks
  - Optimization, Backpropagation, & Learning Rules
- Day 5: Deep Learning (Neural Networks Cont.)
  - Brief Background
  - Convolution Networks
  - Tensorflow & Keras

Plan for the Week

- Day 1: Intro & Review
  - Introduction & Course Overview
  - Review of Fundamentals for Machine Learning in Python
- Day 2: Intro to Classification Algorithms
  - Probability Theory
  - Common & Probabilistic Classifiers
- Day 3: Tree-Based Methods for Classification
  - Decision Tree
  - Random Forest
- Day 4: Neural Networks
  - Perceptron (Single & Multilayer) & ADALINE Networks
  - Optimization, Backpropagation, & Learning Rules
- Day 5: Deep Learning (Neural Networks Cont.)
  - Brief Background
  - Convolution Networks
  - Tensorflow & Keras

What is Machine Learning?

- "A field of Computer Science that gives computers the ability to learn without being explicitly programmed."
  - Arthur Samuel (Coined the term in 1959 at IBM)
- "The ability [for systems] to acquire their own knowledge, by extracting patterns from raw data."
  - *Deep Learning*, Goodfellow et al
- "A computer program is said to learn from experience E with respect to some set of tasks T and performance measure P if its performance tasks in T, as measured by P, improves with experience E."
  - Tom Mitchell (Computer Scientist & Professor at Carnegie Mellon)

Types of Machine Learning

- There are *three* main types of Machine Learning:
  1. Supervised
  2. Unsupervised
  3. Reinforcement

Types of Machine Learning

- There are *three* main types of Machine Learning:
  1. Supervised
  2. Unsupervised
  3. Reinforcement

Supervised Learning

- For each input, there is a target value (label) associated with it
  - Know the right answer ahead of time
- Classification and Regression are subsets of supervised learning
  - This course will focus specifically on *classification*
- Common Examples
  - Classifying Fraudulent Credit Card Transactions
  - Identifying tumors within X-ray images
  - Predicting the price of a house given its characteristics

Types of Machine Learning

- There are *three* main types of Machine Learning:
  1. Supervised
  2. Unsupervised
  3. Reinforcement

Unsupervised Learning

- For each input, there is *no* target value (label) associated with it
  - Do not know the right answer
  - No error to evaluate in our model
- Examples of unsupervised learning
  - Clustering (Customer Segmentation)
  - Dimensionality Reduction (Principal Components Analysis)
  - Anomaly Detection (Outlier Analysis)

Types of Machine Learning

- There are *three* main types of Machine Learning:
  1. Supervised
  2. Unsupervised
  3. Reinforcement

Reinforcement Learning

- For each input, there is *no* target value (label) associated with it
- Area of ML inspired by behavioral psychology
  - Concerned with how agents should act in their environment so as to maximize some cumulative reward.
- Examples of Reinforcement Learning
  - Self-driving car
  - Chess bot

Programming Languages

- Need framework to implement Machine Learning algorithms
- There are *hundreds* of programming languages
  - https://en.wikipedia.org/wiki/List_of_programming_languages
- This course will use Python

Why Python?

- There are several benefits to using python for Machine Learning
  - **Interpreted** Language
    - No need to compile & easier to implement
  - **Dynamic** Typing
    - No need to declare variable types and then assign them
  - **Strong** Typing
    - Once a value is initialized and saved, it is never coerced. Type casting must be done explicitly
  - **High Level**
    - More human readable (white space matters)
  - Machine Learning & Deep Learning **API**s
    - Scikit-Learn, Tensorflow, Keras, CNTK, Apache MXNET, PyTorch, etc.

Why Do We Need Math?

- There's lots of ML code out there that you can just download and run
  - To get useful insights & meaningful results, you need to understand the mathematical underpinnings
  - Knowing the algorithms helps you know when to choose each one for a given problem
  - Recognize over/underfitting
  - Troubleshooting poor/ambiguous results

Software & Tech Stack

The software and frameworks we will be using are as follows:

- Language
  - Python 3.5 (Anaconda Distribution)
- ML/DL Frameworks
  - scikit-learn & Keras
- IDE
  - Spyder
- Course Content Delivery
  - Github
- Comunication
  - Slack

Setting up Slack

- Invite Link:
  - https://join.slack.com/t/hpsi-classification/shared_invite/enQtMjk5MjQzMTMxNzM1LTA2NDU2NTcyZGVjZDlkZThhNjZmN2MwYThmNWI2ZTZlYTI2OTQyOTkyN2M4MmE4OWZkMzFmNzY5MWJmYjZjNTc

Getting Access to the Course Content on GitHub

- If you do not have a GitHub account, please go to the following link and make one:
  - https://github.com/join?source=header-home

- Send your GitHub username to us through the **#github-usernames** channel on Slack
  - You should receive an email inviting you to collaborate to the *HPSI-Classification* repository
  - Clone the repository
  - Move the repository from your `Downloads` folder to your `Desktop` or `Documents` folder

Tech Stack Setup & Testing of Working Environment

- Follow the instructions in the `README.md` file to:
  - Setup the conda environment
    - To learn more about managing conda environments go to: https://conda.io/docs/user-guide/tasks/manage-environments.html#creating-an-environment-from-an-environment-yml-file (Optional)
  - Activate the conda environment
  - Run the `import_test.py` script to ensure your environment has been configured properly

Python Concepts to Know

- Base Python (Assumed)
  - Data Types
    - Booleans, numbers (int/float), strings, lists, tuples, sets, dictionaries
  - Conditionals
    - if/elif/else
  - Loops
    - for/while
  - Functions
- Python Cheat Sheets

Math Review

- Next we are going to review some math we will need for the algorithms and topics discussed throughout the rest of the course
- Machine Learning is primarily based on
  - Linear Algebra
  - Calculus & Optimization
  - Probability Theory

Linear Algebra

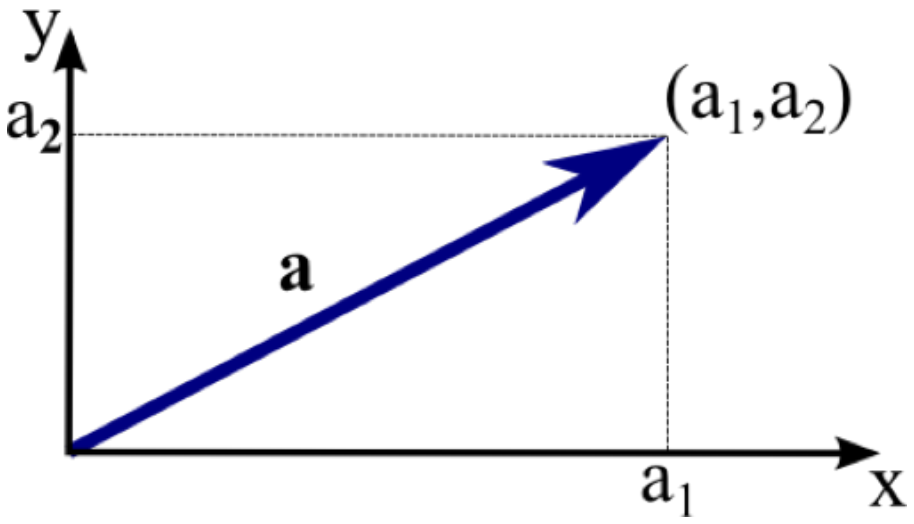
- The study of Linear Algebra involves the following types of objects:
  - Scalars
  - Vectors
  - Matrices
  - Tensors

Linear Algebra

- A **scalar** is just a single number
  - Quantity that only has *magnitude*
  - Usually denoted by a lowercase, italicized letter such as *c*

```
# scalar addition
>>> a = 2
>>> b = 4
>>> a + b
6
```

- A **vector** is an 1-D array of numbers
  - Has both *magnitude* and *direction*
  - Usually denoted by a lowercase, bold letter such as **x**
  - Written as a *column* enclosed in square brackets
    - $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$
    - The dimension of the vector is given by *n*
    - Each element of the vector is identified by an index
      - For example, x_2 represents the second element
  - We can think of vectors as identifying points in space
    - Each element gives the coordinate of a different axis



Linear Algebra

```
# vector is 1-D array
>>> import numpy as np
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])

# check number of dimensions
>>> a.ndim
1

# check the number of rows and columns
>>> a.shape
(4,)
```

Linear Algebra

- A **matrix** is a 2-D array of numbers
  - Usually denoted by an uppercase, bold letter such as ***A***
  - Often thought of as an array of column vectors
  - Identified by *two* indices
    - Has *m* rows and *n* columns
    - $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_{1,1}} & \cdots & \boldsymbol{A_{1,n}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{A_{m,1}} & \cdots & \boldsymbol{A_{m,n}} \end{bmatrix}$

Linear Algebra

```
# matrix is 2-D array
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])

# check number of dimensions
>>> a.ndim
2

# check the number of rows and columns
>>> a.shape
(2, 2)
```

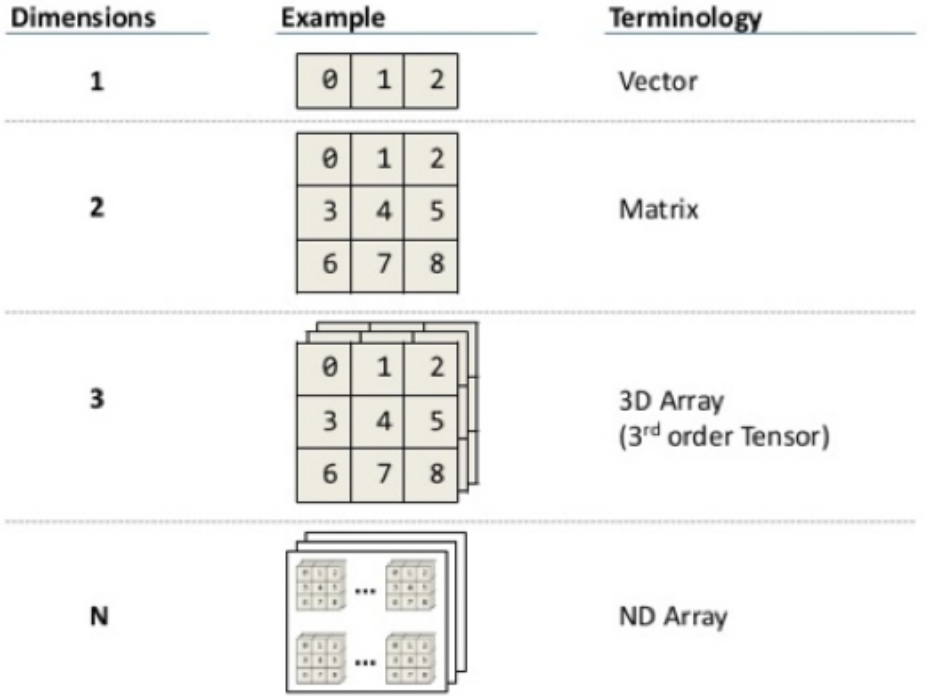
Linear Algebra

- A **tensor** is a generalization of vectors and matrices to higher dimensions

```
# tensor is higher dimensional array
>>> a.reshape(1, 2, 2)
array([[[0, 1],
        [2, 3]]])

>>> a.reshape(1, 2, 2).shape
(1, 2, 2)

>>> a.reshape(1, 2, 2).ndim
3
```



Linear Algebra

- Transpose of vectors and matrices
  - The **transpose** of a vector or matrix is when we reflect it across the main diagonal
    - More simply: Swap the rows and columns

Linear Algebra

```
# transposing vectors
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a.shape
(5,)

# method to transpose
>>> a.T
array([0, 1, 2, 3, 4])
>>> a.T.shape
(5,)

# function to transpose
>>> np.transpose(a)
array([0, 1, 2, 3, 4])
>>> np.transpose(a).shape
(5,)
```

- Why didn't anything happen?

Linear Algebra

```
# transposing vectors continued
>>> a.reshape(-1,1)
array([[0],
       [1],
       [2],
       [3],
       [4]])
>>> a.reshape(-1,1).shape
(5, 1)

>>> a.reshape(-1,1).T
array([[0, 1, 2, 3, 4]])
>>> a.reshape(-1,1).T.shape
(1, 5)
```

Linear Algebra

```
# transposing matrices

>>> b = np.arange(9).reshape(3,3)
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> b.T
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])

>>> b.transpose()
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
```

Linear Algebra

- Vector addition
  - Vectors must have the same length
  - Addition is done element-wise
    - $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y} = (\mathrm{x}_1 + y_1 \, , \cdots , \mathrm{x}_n + y_n)^T$
    - Result is another vector

```
# vector addition

>>> a = np.arange(6)
>>> b = a + 4

>>> a
array([0, 1, 2, 3, 4, 5])
>>> b
array([4, 5, 6, 7, 8, 9])

>>> a + b
array([ 4,  6,  8, 10, 12, 14])
```

Linear Algebra

- Matrix arithmetic
  - If ***A*** and ***B*** are the same size (same number of rows and columns), the resulting matrix has the same size
- $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B} \implies \mathrm{C}_{i,j} = \mathrm{A}_{i,j} + B_{i,j}$

```
# matrix addition

>>> a = np.arange(4).reshape(2,2)
>>> b = a + 4

>>> a
array([[0, 1],
       [2, 3]])
>>> b
array([[4, 5],
       [6, 7]])
>>> a + b
array([[ 4,  6],
       [ 8, 10]])
```

Linear Algebra

- The **dot product** of two vectors
  - Results in a *scalar*
  - Multiply corresponding elements, then add the products
  - $\boldsymbol{a} = \boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i$

```
>>> a = np.arange(6)
>>> b = a + 4

>>> a
array([0, 1, 2, 3, 4, 5])
>>> b
array([4, 5, 6, 7, 8, 9])

# element-wise multiplication
>>> a*b
array([ 0,  5, 12, 21, 32, 45])

# dot product
>>> a.dot(b)
115
```

Linear Algebra

- The **norm** of a vector ***x*** is the length from the origin to ***x***
  - Used for measuring the size of a vector
  - Maps vectors to non-negative values
- L^p Norm
  - $\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$
- L^2 Norm (Euclidean)
  - $\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2\right)^{1/2} = \sqrt{\sum_{i=1}^{n} |x_i|^2}$
- L^1 Norm (Manhattan/Taxicab)
  - $\|x\|_1 = \left(\sum_{i=1}^{n} |x_i|^1\right)^{1/1} = \sum_{i=1}^{n} |x_i|$
- We will talk more about why you would want to use different distance metrics later on in day 2

Linear Algebra

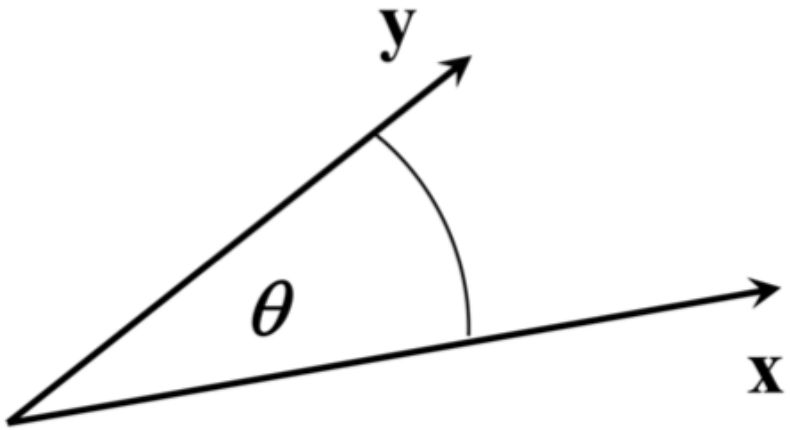
```
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])

# euclidean norm
>>> np.linalg.norm(a)
7.416198487095663

# taxicab norm
>>> np.linalg.norm(a, ord=1)
15.0
```

- Alternative representation of the dot product
  - $\boldsymbol{a} = \boldsymbol{x} \cdot \boldsymbol{y} = \|\boldsymbol{x}\|\|\boldsymbol{y}\| \cos\theta$

- Recall
  - $\cos 0° = 1$
  - $\cos 90° = 0$



- The dot product equals 0 when $\boldsymbol{x} \perp \mathbf{y}$
- The dot product equals 1 when $\overleftrightarrow{\boldsymbol{xy}}$ (collinear)

Linear Algebra

- Matrix multiplication
  - The inner dimension of the matrices have to match
    - In a chain of matrix multiplications, the *column* dimensions must match the *row* dimensions of the following matrix in the chain
  - The resulting size is the number of *rows* of the first and the number of *columns* of the second

Linear Algebra

```
>>> a = np.arange(4).reshape(2,2)
>>> b = a + 4

>>> a
array([[0, 1],
       [2, 3]])
>>> b
array([[4, 5],
       [6, 7]])

# element-wise multiplication
>>> a*b
array([[ 0,  5],
       [12, 21]])

# matrix multiplication
>>> a.dot(b)
>>> a@b
>>> np.matmul(a,b)
array([[ 6,  7],
[26, 31]])
```

Linear Algebra

- Matrix multiplication
  - Example matrices
    - **A** is 3x5
    - **B** is 5x1
    - **C** is 5x2
  - Can we compute?
    - **AB**
    - **AC**
    - **BC**
    - **BA**
    - **CA**

Linear Algebra

- Matrix multiplication
  - Example matrices
    - **A** is 3x5
    - **B** is 5x1
    - **C** is 5x2
  - Can we compute?
    - **AB** ✅
    - **AC** ✅
    - **BC** ❌
    - **BA** ❌
    - **CA** ❌

Machine Learning Overview

There are several Python ML frameworks

This course will begin by using scikit-learn

- One of the most popular and user-friendly machine learning libraries
- Plays nice with Numpy, Pandas, & SciPy

Other ML/DL libraries to be aware of:

- Keras
  - TensorFlow
  - Theano
- Torch/Pytorch
- Caffe/Caffe2
- CNTK
- MXNet
- SparkML

Machine Learning Overview

- scikit-learn is a python package that provides efficient implementations of most common ML algorithms via a simple API interface
- scikit-learn has great online documentation
  - http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html
- Uniform syntax for “Estimators”
  - Once you know how to code one model, you can code them all

Machine Learning Overview

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Machine Learning Overview

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Define the Problem

- What are we classifying?
  - Identify the target variable of interest
  - Decide how that variable ought to be measured
- Choose a *performance metric* based on the goal
  - Accuracy is the simplest classification metric, but it may not be appropriate depending on the application
  - We want to find the metric closest to our business goal that is feasible to evaluate
  - We'll circle back to performance metrics in model evaluation

Machine Learning Overview

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Machine Learning Overview

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Explore the Data

- How big is the dataset?
- What types of features do we have?
- Do we have missing data?
  - Do we know why that data is missing?
- How are the data distributed?
  - Class breakdown & Visualization

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Preprocess the Data

- *Imputing* Missing Values
  - Method: mean, median, mode, interpolation, etc.
  - Sometimes makes sense to drop observation if it's missing several measurements
    - Drop rows based on threshold criterion
- *Encoding* Categorical Features
  - Method determined by whether feature is *nominal* or *ordinal*
- *Structuring* Data for Modeling in scikit-learn
- *Scaling* features
  - Normalization/Standardization
- *Engineering* Features
  - Not explicitly covered in this course
  - This is where domain knowledge comes in hand

Dealing with Missing Data

- What causes missing data?
  - Error in data collection
  - Certain measurements are “Not-Applicable”
  - Fields may have been left blank during data entry
- Null values
  - NaN vs “ ” vs “_” vs “.” vs “N/A” vs “N-A”
- What to do?
  - Impute
    - Mean
    - Median
    - Most Frequent Class
    - Forward-Fill
    - Back-Fill
  - Drop based on a missing-value threshold

Quantifying Missing Data

- Count the number of missing values per column:

```
>>> df.isnull().sum()
```

- Simply remove the corresponding samples from the dataset entirely:

```
# axis = 0 for rows
# axis = 1 for columns
>>> df.dropna(axis=0)

#only drop rows where all columns are NaN
>>> df.dropna(how = 'all')
```

Handling Missing Values

```
# drop rows that have less than 4 real values
>>> df.dropna(thresh = 4)


#only drop rows where NaN appear in specific
columns (here: 'C')
>>> df.dropna(subset = ['C'])
```

Imputing Missing Values

- Usually we have too much valuable data to simply drop columns/rows, in this case we can fill the missing data ("imputation")

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values = 'NaN',
                  strategy = 'mean',
                  axis = 0)
>>> imr.fit(df.values)
>>> imputed_data = imr.transform(df.values)
```

Imputing Missing Values

- Here, we replaced each “NaN” value with the corresponding mean, which is separately calculated for each feature column.
- You can impute based on row means by setting `axis = 1`
- Other imputation strategies that are available include setting the `strategy` parameter to:
  - `median`
    - replaces missing values with the median value
  - `most_frequent` (mode)
    - replaces missing values with the most frequent value in the column

Encoding Categorical Features

- Nominal
  - Categorical features that don't imply an order
    - T-shirt colors (Red, Blue, Green)
    - Animals (Dog, Cat, Fish, Tiger)
- Ordinal
  - Categorical features that can be sorted or have an order
    - T-shirt size (S, M, L)
    - Quality (Poor, Fair, Good, Excellent)

Encoding Categorical Features

- Let's make a dummy dataset to walk through encoding categorical features

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...             ['green', 'M', 10.1, 'class1'],
...             ['red', 'L', 13.5, 'class2'],
...             ['blue', 'XL', 15.3, 'class1']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

Encoding Categorical Features

- Mapping Ordinal Features
  - To ensure the algorithm interprets the order appropriately, we need to convert the categorical string values into integers.
  - There is no function to automatically derive the correct order of the labels of our size feature, so we have to define the mapping manually
  - Assume the relation XL = L + 1 = M + 2 holds

Encoding Categorical Features

➢Mapping Ordinal Features

```
>>> size_mapping = {
...                 'XL': 3,
...                 'L': 2,
...                 'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
      color  size  price classlabel
0     green     1   10.1     class1
1       red     2   13.5     class2
2      blue     3   15.3     class1
```

HPSI
Human Performance Systems, Inc.

Encoding Categorical Features

Inverse Mapping Ordinal Features

- If want to recover the original string representation later on, we can define a reverse-mapping dictionary:

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
```

  - can use with pandas `map()` method on the transformed feature column, similar to the previous mapping

```
>>> inv_size_mapping = {v: k for k, v in
size_mapping.items()}
>>> df['size'].map(inv_size_mapping)
0      M
1      L
2     XL
Name: size, dtype: object
```

Encoding Class Labels

- Most sklearn classification estimators convert class labels to integers internally
- Still good practice to provide class labels as integer arrays to avoid issues
- Class labels are not ordinal
- It doesn't matter which integer number we assign to a particular string label.
  - Thus, we can simply enumerate the class labels, starting at 0

Encoding Class Labels

- The `LabelEncoder` class in sklearn does this for us:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

Encoding Class Labels

- The `LabelEncoder` class in sklearn does this for us:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

Encoding Class Labels

- Note that the `fit_transform` method is just a shortcut for calling `fit` and `transform` separately
  - `fit` makes the mapping
  - `transform` applies the mapping
  - we can use the `inverse_transform` method to transform the integer class labels back into their original string representation

```
>>> class_le.inverse_transform(y)
array(['class1', 'class2', 'class1'], dtype=object)
```

Performing One-Hot Encoding on Nominal Features

- Sklearn treats class labels as nominal variables, so we were able to use the `LabelEncoder` to encode the strings as integers
- We won't be able to use this method for the nominal features in our data though

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

Performing One-Hot Encoding on Nominal Features

- The new values for the *color* feature are as follows:
  - Blue = 0
  - Green = 1
  - Red = 2
- Although the color values don't come in any particular order, the classifier will assume that green is larger than blue, and red is larger than green
- The solution: **One-hot encoding**
  - Create a new dummy feature for each unique value in the nominal feature column

Performing One-Hot Encoding on Nominal Features

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

- Specify columns to one-hot encode when you instantiate
- The `OneHotEncoder` returns a sparse matrix by default
  - We use `toarray` to see it
  - Could also instantiate `OneHotEncoder` with the `sparse=False` parameter

Performing One-Hot Encoding on Nominal Features

- We can also do this with pandas:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0   10.1     1           0            1          0
1   13.5     2           0            0          1
2   15.3     3           1            0          0
```

Caution with One-Hot Encoding

- One-Hot Encoding introduces multicollinearity to our dataset
  - Not great for methods that rely on matrix inversion
- To reduce multicollinearity, we can drop columns that don't give much additional info
  - If we remove the column `color_blue`, the feature information is still preserved since if we observe `color_green=0` and `color_red=0`, it implies that the observation is blue

```
>>> pd.get_dummies(df[['price', 'color', 'size']],
...                drop_first=True)
   price  size  color_green  color_red
0   10.1     1            1          0
1   13.5     2            0          1
2   15.3     3            0          0
```

Structuring Data for Modeling in Sklearn

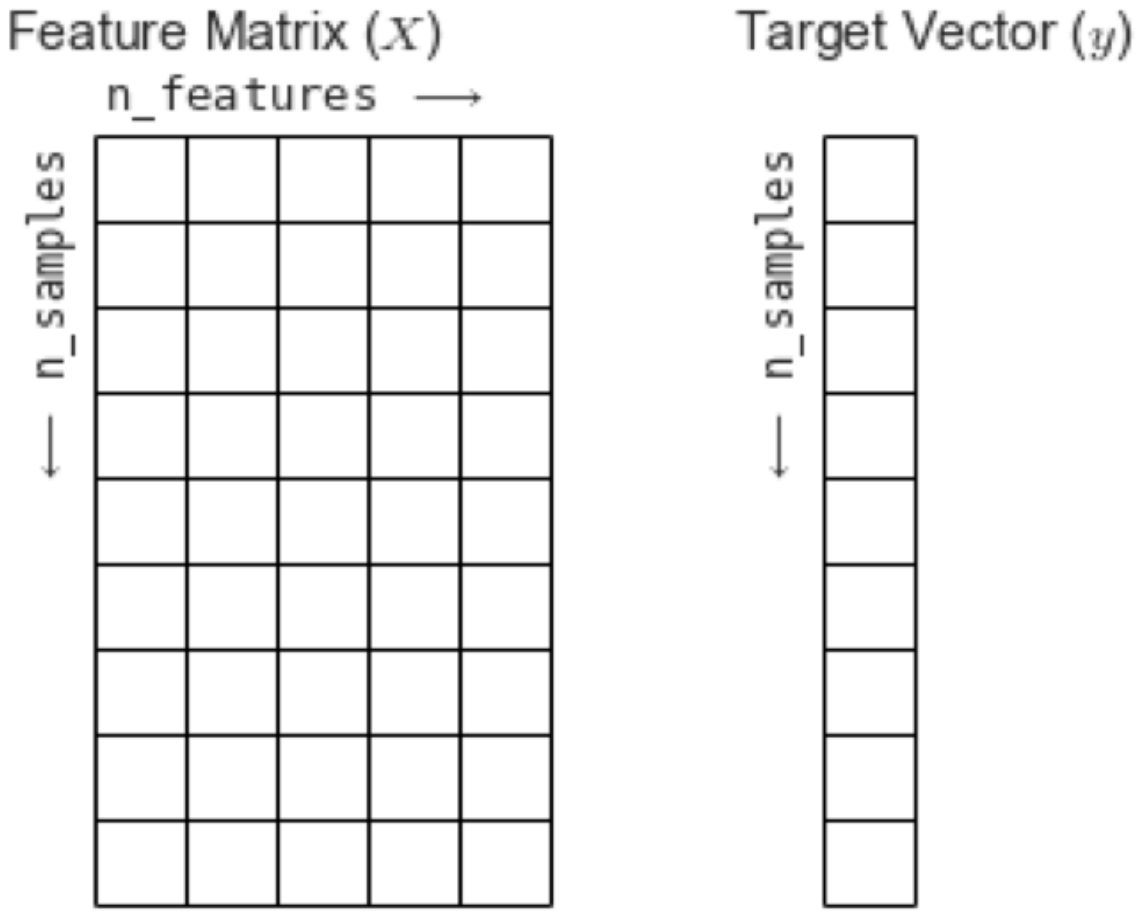
➢Feature matrix (X)

- ❖Scikit-learn assumes a 2-D array of inputs with dimensions [`n_samples`, `n_features`]
- ❖Scikit-learn expects a feature matrix as a NumPy array or Pandas DataFrame
  - ➢ some Scikit-Learn models also accept SciPy sparse matrices.

➢Target vector (y)

- ❖Usually a column vector of length `n_samples`
- ❖scikit-learn expects a NumPy array or Pandas Series.

Structuring Data for Modeling in Sklearn



Structuring Data for Modeling in Sklearn

- We've split our data up into a feature matrix & target vector, but we still aren't ready to model it yet
  - If we fit our model to all the data and test the model on all the data, we would correctly predict almost every sample in the training set
    - Memorizing; not learning
  - Want our models to *generalize*
    - Perform well on unseen (out-of-sample) data
  - Holdout some data to test on
- Split our data into a *training set* and *test set*
  - Learn parameters from the training set and predict/score on the test set

Structuring Data for Modeling in Sklearn

- Sklearn's `train_test_split` function does this for us
  - Its default parameter is to extract 75% for training and 25% for testing

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test =
train_test_split(iris['data'], iris['target'], random_state=0)
```

Splitting into Training and Test Data

- The `train_test_split` function shuffles the data with a pseudorandom number generator before splitting
  - Without shuffling the iris data, we would only have class 2 in the test set
  - To make sure we get the same splits, we set the seed with the `random_state` parameter
- Dealing with imbalanced classes (credit fraud/cancer examples)
  - When dealing with imbalanced datasets, it is important to ensure the training and test set have the same proportions as the original dataset
  - We can do this by passing the target vector as an argument to the `stratify` parameter

```
train_test_split(iris['data'], iris['target'],
random_state=0, stratify=y)
```

Scaling Features

- Some algorithms are very sensitive to how the data are scaled
  - Neural networks → Vanishing gradient
- There are *four* different ways to scale our data in sklearn:
  - `StandardScaler`
  - `RobustScaler`
  - `MinMaxScaler`
  - `Normalizer`

Scaling Features

- `StandardScaler`
  - Scales each feature so that the mean is zero and the standard deviation is one
    - $x_{std} = \frac{x - \mu_x}{\sigma_x}$
    - All features on same magnitude
    - Does not ensure any particular minimum or maximum values
- `RobustScaler`
  - Similar to the `StandardScaler` in that it guarantees features are on the same scale, but uses the median and quartiles to scale instead of mean and variance
    - $x_{robust} = \frac{x - Q_2}{Q_3 - Q_1}, where\ Q_1, Q_2, Q_3\ are\ the\ 1st, 2nd, and\ 3rd\ quartiles, respectively$
    - Makes the scaler *robust* to outliers

Scaling Features

- MinMaxScaler
  - $x_{minmax} = \frac{x - x_{min}}{x_{max} - x_{min}}$
  - Shifts the data such that all features are between zero and one
  - Min-Max scaling is useful when we need values in a bounded interval
  - Standardization can be more practical for many machine learning algorithms especially for optimization algorithms such as gradient descent.
    - Many linear models initialize weights to 0 or small random values close to 0
    - Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns takes the form of a normal distribution
    - Standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

Scaling Features

- `Normalizer`
  - $x_{norm} = \frac{x}{\|x\|}$
  - Scales each point such that the feature vector has a Euclidean length of one
  - Projects data point on unit circle (or *n*-dimensional counterpart)
    - Every point is scaled by a different number (inverse of it's length)
  - Used when only the direction/angle of the data matters, not length

Scaling Example

- Open up the `feature-scaling.py` script to see the different scaling methods in action

Feature Engineering & Feature Selection

- *Feature Engineering* is the process of determining how to best represent your data for a particular application
  - Some would consider encoding categorical features to be part of this process
    - Encoding categorical features vs discretizing continuous variables/interaction terms/polynomial features/transformations
  - Case-by-case and model-dependent
    - No need to scale features for tree-based methods
    - Very important to scale features for *k*-NN & neural networks
  - Industry knowledge is very helpful at this stage

Feature Engineering & Feature Selection

- Encoding and engineering features can cause us to increase the dimensionality of the data greatly beyond the original number of features
  - When adding new features in this way, or with high-dimensional datasets in general, it's a good idea to reduce the number of features to only the most useful ones
  - This can lead to simpler models that *generalize* better
- *Feature Selection* is the process of determining which features to include in our model
  - *Three* main methods:
    1. Univariate Statistics (ANOVA)
    2. Model-based Feature Selection (Trees & LASSO)
    3. Iterative Feature Selection (RFE)

Machine Learning Overview

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Scikit-learn Estimator – Steps to Fit a Model & Generate Predictions

1. **Import** appropriate estimator from scikit-learn
2. **Instantiate** estimator with desired hyperparameters
3. **Fit** the model by calling the `fit()` method on the training set
   - ❖ What's really happening here?
     - ➢ The model is *learning* the relationship between the features/inputs and the response/target
     - ➢ Note: All learned parameters are stored as names with trailing underscores
4. **Predict** on unseen data (test set):
   - ❖ Predict labels with `predict()` method

Scikit-learn Estimator – Steps to Fit a Model & Generate Predictions

```
# import sklearn methods and estimator
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.neighbors import KNeighborsClassifier
>>> iris = load_iris()
# feature matrix and target vector
>>> X = iris.data
>>> y = iris.target
# split into training and test set
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=0)
# instantiate estimator
>>> knn = KNeighborsClassifier(n_neighbors=1)
# fit on training set to 'learn' parameters
>>> knn.fit(X_train, y_train)
# predict on the test set
>>> y_pred = knn.predict(X_test)
```

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Performance Metrics for Binary Classification Models

- In *binary* classification, we have 2 classes—positive and negative
- **Accuracy:** Fraction of *correctly* classified samples
  - number correctly classified / total number of samples
  - (TP+TN)/total
    - A *True Positive* is when the sample is from the *positive* class and we *correctly* predict the *positive* class
    - A *True Negative* is when the sample is from the *negative* class and we *correctly* predict the *negative* class
    - A *False Positive* is when the sample is from the *negative* class and we *incorrectly* predict the *positive* class (Type I Error)
    - A *False Negative* is when the sample is from the *positive* class and we *incorrectly* predict the *negative* class (Type II Error)

Performance Metrics for Binary Classification Models

- Each estimator in scikit-learn has a `score` method that computes the default evaluation criterion for the problem they are designed to solve
  - For almost all classifiers, this will be accuracy
- In our previous example, the following command would calculate the accuracy of the classifier:

```
knn.score(X_test, y_test)
0.9736842105263582
```

- There are also explicit functions in scikit-learn for all the performance metrics

Accuracy in sklearn

```
>>> import numpy as np
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]

# fraction of classified samples
>>> accuracy_score(y_true, y_pred)
0.5

# number of correctly classified samples
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

When Would You Not Want to Use Accuracy?

- When the number of mistakes does not contain all the information we are interested in
- **Example:** Imagine an automated application to screen for the early detection of cancer. If the test is *negative*, we assume the patient is healthy, and if the test is *positive*, the patient will undergo additional screening.
  - Positive Test → Cancer
  - Negative Test → Healthy
- Our model is never going to be going to be perfect

When Would You Not Want to Use Accuracy?

- What are the consequences of being wrong?
  1. Healthy patient is told they have cancer (False Positive)
     - Costs and inconvenience to patient
  2. Sick patient is told they are healthy (False Negative)
     - Patient will not receive further tests & treatment
     - Undiagnosed cancer could lead to serious issues or ☠
- We obviously want to avoid *False Negatives* as much as possible in this example

When Would You Not Want to Use Accuracy?

- We run into the same issue with imbalanced datasets

- **Example:** Imagine we are looking at credit fraud data where 99% of the transactions were not fraudulent and 1% were fraudulent
  - Say we build a complex ML model that has 99% accuracy in predicting whether a transaction is fraudulent or not
    - Is this actually good?
      - We can just guess every single time that every transaction is not fraudulent and get an accuracy of 99%
      - We actually don't know if the model is good because accuracy doesn't give us enough information as to how predictions were made

Confusion Matrix

- The *confusion matrix* is a more informative way to represent the results of a binary classifier

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Confusion Matrix

- 165 total observations/predictions

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Confusion Matrix

- Of the 165 total observations, we predicted *yes* 110 times

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Confusion Matrix

- Of the 165 total observations, we predicted *no* 55 times

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Confusion Matrix

- Of the 165 total observations, 105 are actually *yes*

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Confusion Matrix

- Of the 165 total observations, 60 are actually *no*

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Confusion Matrix

- Diagonal terms are correct classifications

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Confusion Matrix

- Accuracy = (TN + TP) / total = (100 + 50) / 165 = 0.91

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Confusion Matrix in sklearn

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [0, 1, 0, 1]
>>> y_pred = [1, 1, 1, 0]
>>> confusion_matrix(y_true, y_pred)
array([[0, 2],
       [1, 1]])

# we can extract the following info in the binary case
>>> tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
>>> print((tn, fp, fn, tp))
(0, 2, 1, 1)
```

Metrics Derived from Confusion Matrix

- **Misclassification Rate:** Proportion *incorrectly* classified
  - (FP+FN)/total
  - Misclassification rate = 1 – Accuracy
- **Precision:** When it predicts *yes*, how often is it correct?
  - TP/(TP + FP)
    - Bounded between [0,1] where 1 is the best
  - *Used when the goal is to limit the number of false positives*
    - Clinical trials are expensive, so you want to know a drug works before investing in them
- **Recall (True Positive Rate):** When it's actually *yes*, how often does it predict *yes*?
  - TP/(TP + FN)
    - Bounded between [0,1] where 1 is the best
  - *Used when we need to identify all positive samples (avoid false negatives)*
    - *Cancer example*

Confusion Matrix

- Misclassification Rate = (FP+FN) / total = (10 + 5) / 165 = 0.09

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Confusion Matrix

- Precision = TP / (TP + FP) = 100 / (100 + 10) = 0.91

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Confusion Matrix

- Recall = TP / (TP + FN) = 100 / (100 + 5) = 0.95

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

HPSI
Human Performance Systems, Inc.

Misclassification Rate, Precision, & Recall in sklearn

```
>>> from sklearn.metrics import accuracy_score, precision_score,
recall_score

>>> miss_rate = 1 - accuracy_score(y_true, y_pred)
>>> precision = precision_score(y_true, y_pred)
>>> recall = recall_score(y_true,y_pred)

>>> print("Misclassification Rate: {:03.2f}".format(miss_rate))
>>> print("Precision: {:03.2f}".format(precision))
>>> print("Recall: {:03.2f}".format(recall))
Misclassification Rate: 0.75
Precision: 0.33
Recall: 0.50
```

Metrics from Confusion Matrix

- ***f1*-score**
  - Harmonic mean of precision and recall
    - Harmonic mean is used when we want to average rates
  - $2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$
    - Bounded between [0, 1] where 1 is best
    - Equal contribution between precision and recall
    - Hardest to interpret

f1-score in sklearn

```
>>> from sklearn.metrics import f1_score
>>> print("f1-score: {:03.2f}".format(f1_score(y_true,
y_pred)))

# can do precision, recall, and f1 all together
from sklearn.metrics import classification_report
print(classification_report(y_true, y_pred))
                  precision recall f1-score support
              0       0.00   0.00     0.00       2
              1       0.33   0.50     0.40       2
avg / total           0.17   0.25     0.20       4
```

HPSI
Human Performance Systems, Inc.

Performance Metrics for Binary Classification Models

- The confusion matrix and classification report provided much more information about the predictions than accuracy
  - But, can we get even more information?
    - Yes!
  - We can take uncertainty into account too
    - Most classifiers in sklearn have either a *decision_function* or *predict_proba* method
    - Predictions can be seen as ***thresholding*** the output of one of these functions
      - `decision_function` represents the signed difference of an observation to the plane
        - Assigns all points with a value greater than zero to the positive class
      - `predict_proba` represents the predicted probability of belonging to each class
        - Assigns the point to class based on highest probability

Thresholding

- Depending on the end goal, we can adjust the decision threshold for the probability function
- Recall the cancer screening example where we want a high recall for the positive class
  - Willing to risk more *false positives* in order to reduce *false negatives*
  - We might consider *decreasing* the threshold on the predictions in order to *increase* the recall
    - Recall = TP / (TP + FN)
    - Precision = TP / (TP + FP)
    - Decrease FN → Increase Recall
    - Increase FP → Decrease Precision
- Adjusting the threshold is a trade-off between precision and recall
  - **You should NEVER set your threshold based on the test set**
  - **Always use the training or validation set to determine your optimal threshold**

Precision-Recall Curves

- Thresholding is particularly helpful when we have a set precision or recall score determined by the business project
- Once a precision or recall score is set, a threshold can be set
- Most of the time, we do not have a set point and may wish to look at all possible thresholds and trade-offs at once
  - We can do this with the *precision-recall curve*

Precision-Recall Curve in sklearn

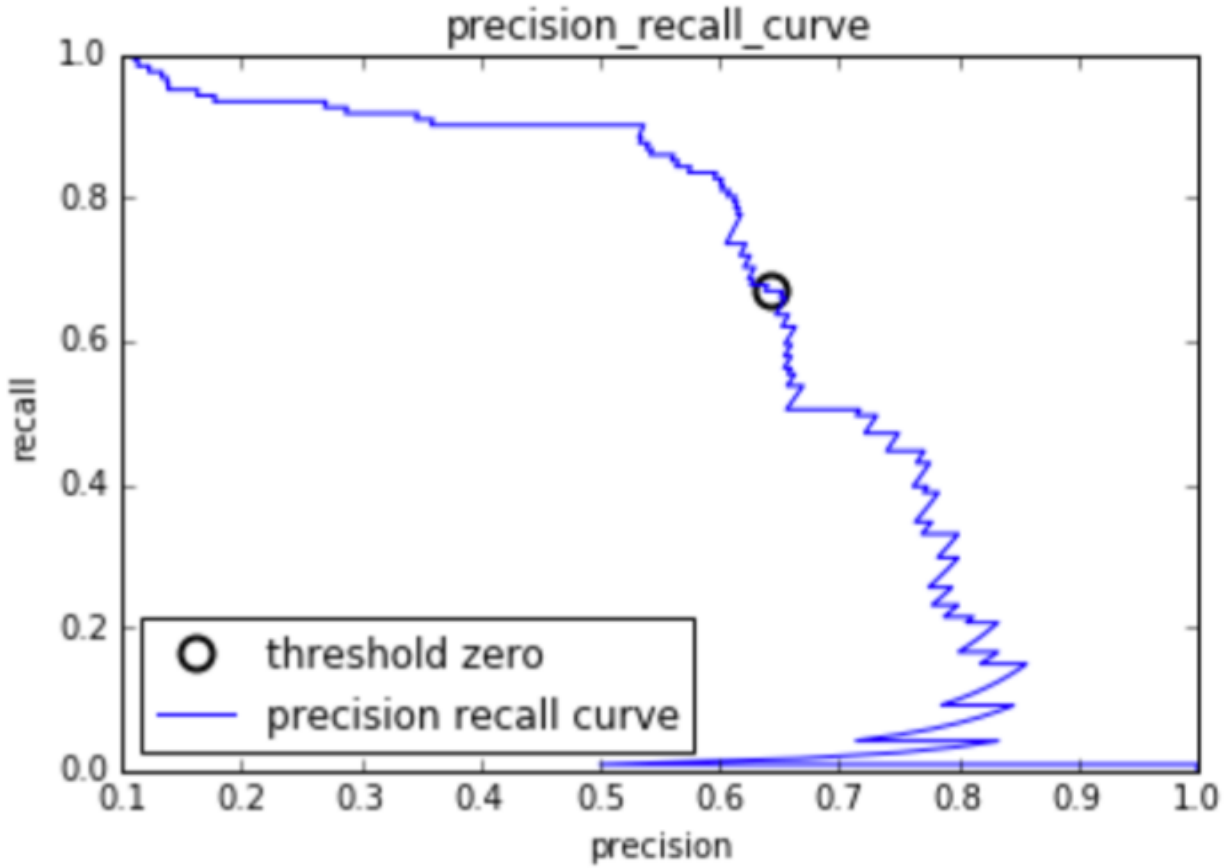
```
>>> from sklearn.metrics import precision_recall_curve
>>> precision,recall,thresholds =
precision_recall_curve(y_test,
                       model.decision_function(X_test))
```

- The `precision_recall_curve` function returns a list of precision and recall values for all possible thresholds (all values that appear in the decision function) in sorted order
- We can use this to plot the curve

Precision-Recall Curve in sklearn

```
# find threshold closest to zero:
>>> close_zero = np.argmin(np.abs(thresholds))
>>> plt.plot(precision[close_zero], recall[close_zero],
'o', markersize=10, label="threshold zero",
fillstyle="none", c='k', mew=2)
>>> plt.plot(precision, recall, label="precision recall
curve")
>>> plt.xlabel("precision")
>>> plt.ylabel("recall")
>>> plt.title("precision_recall_curve")
>>> plt.legend(loc="best")
```

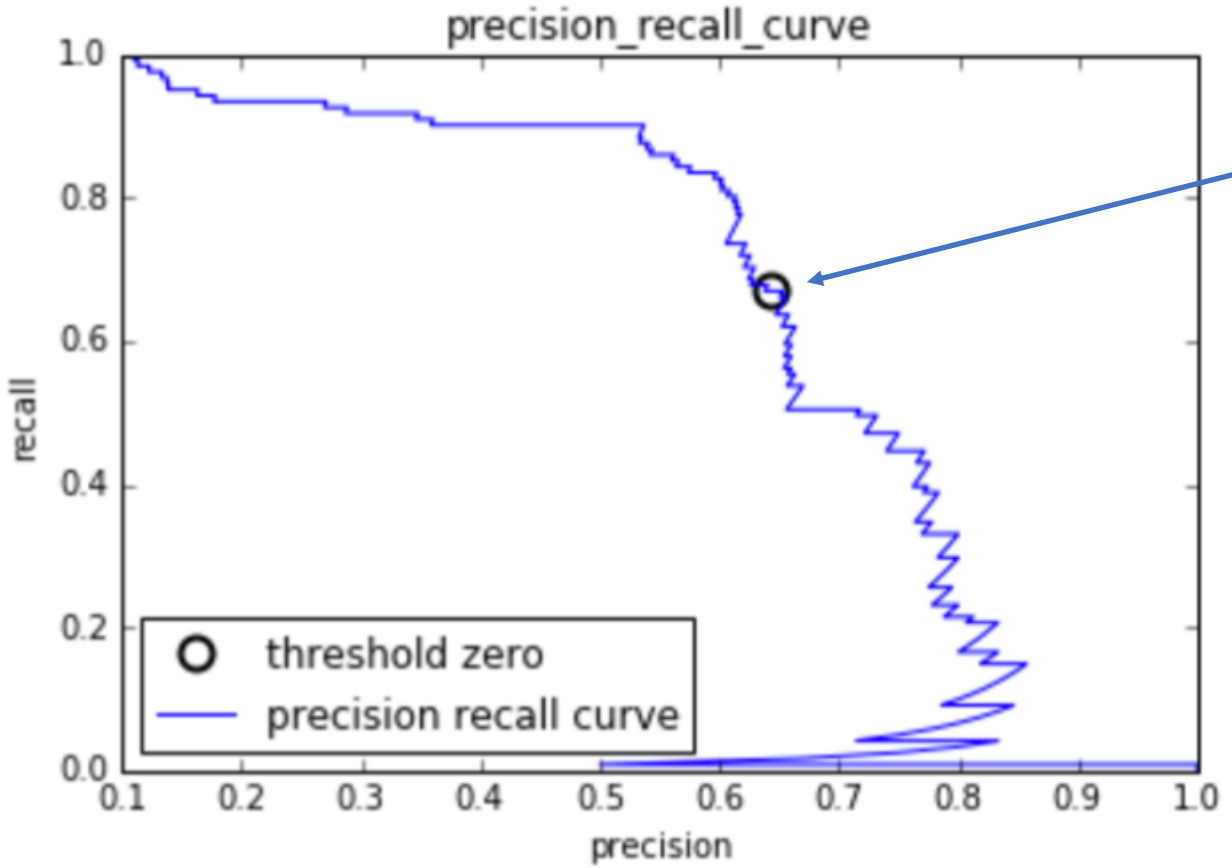
Precision-Recall Curves



Each point along the curve corresponds to a possible threshold from either `predict_proba` or `decision_function`

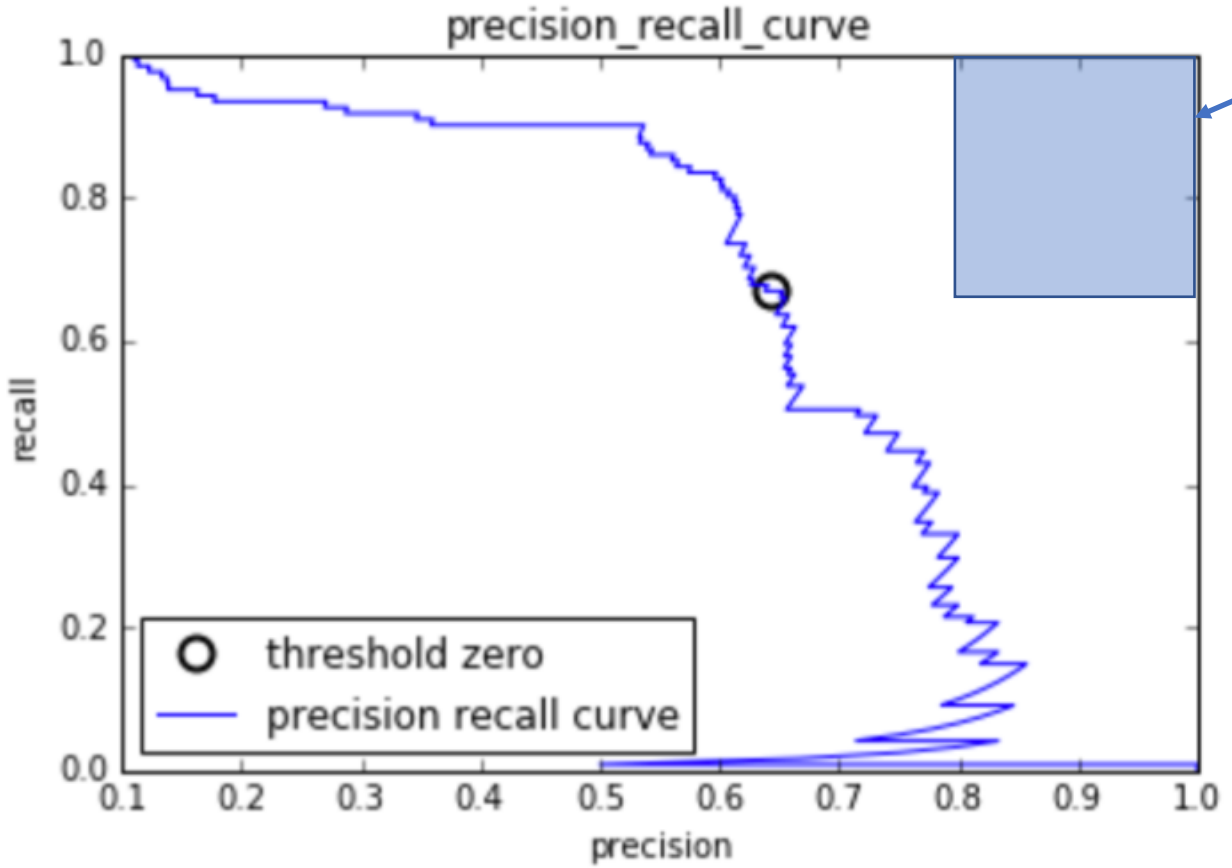
HPSI
Human Performance Systems, Inc.

Precision-Recall Curves



We can write code to find the threshold closest to zero in the list of returned thresholds and plot it to see where the default decision threshold is

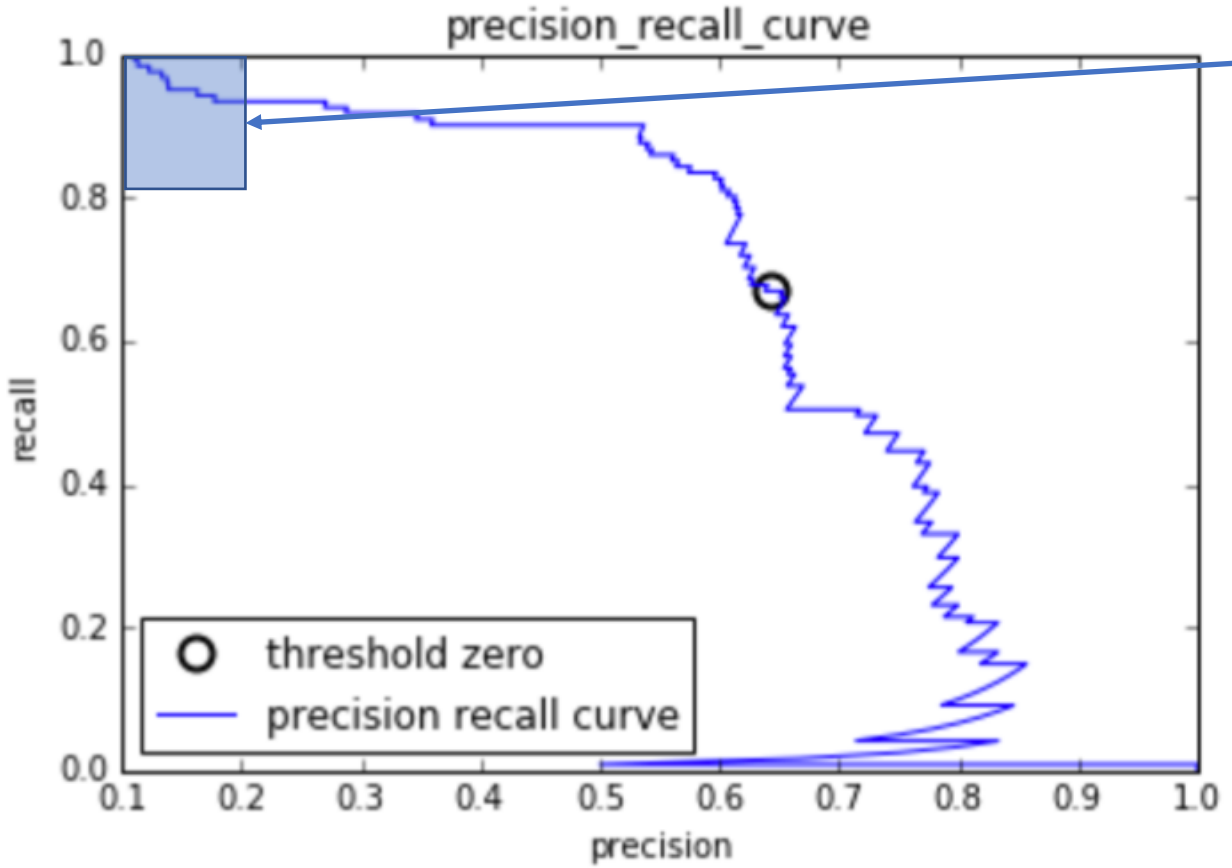
Precision-Recall Curves



The closer a curve stays to the upper-right corner, the better the classifier

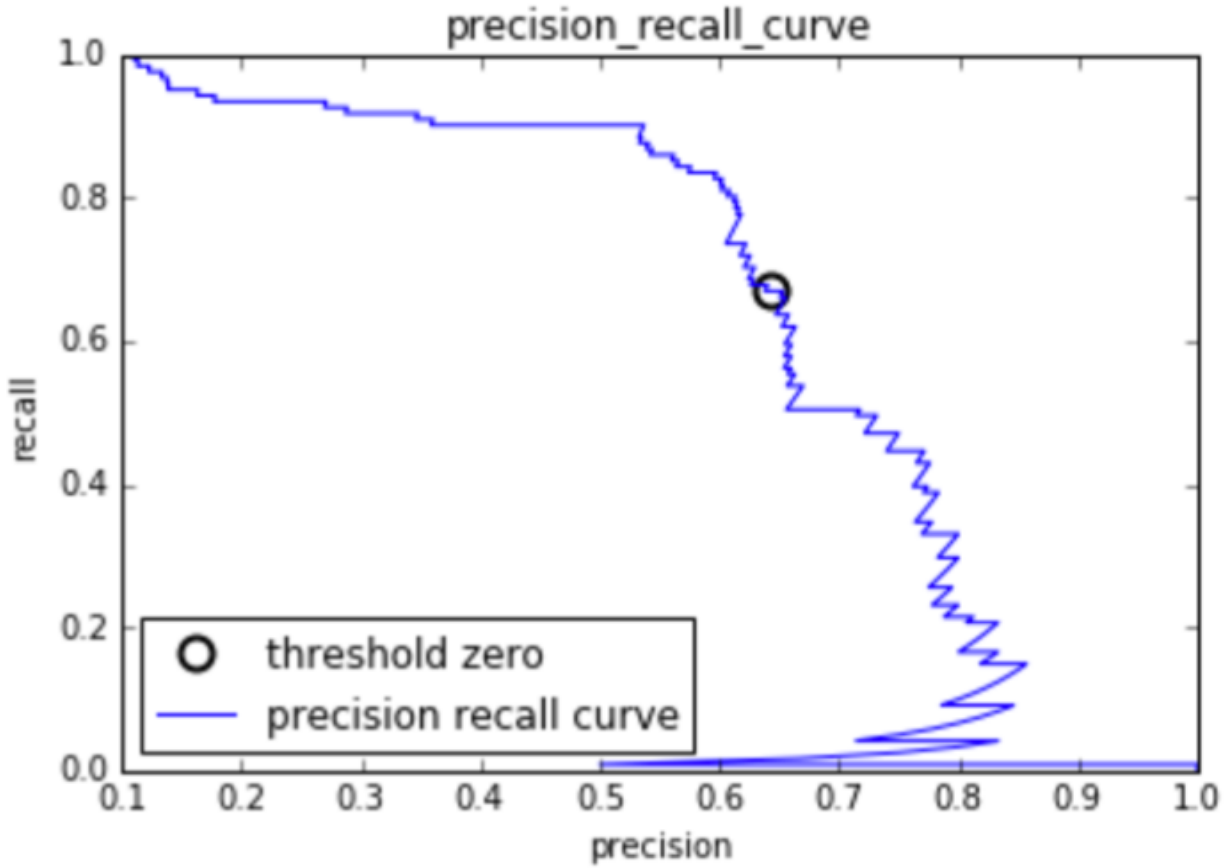
A point in the upper-right means a classifier has high precision & high recall

Precision-Recall Curves



The curve starts in the top-left corner, corresponding to a very low threshold, which classifies everything as the *positive* class

Precision-Recall Curves

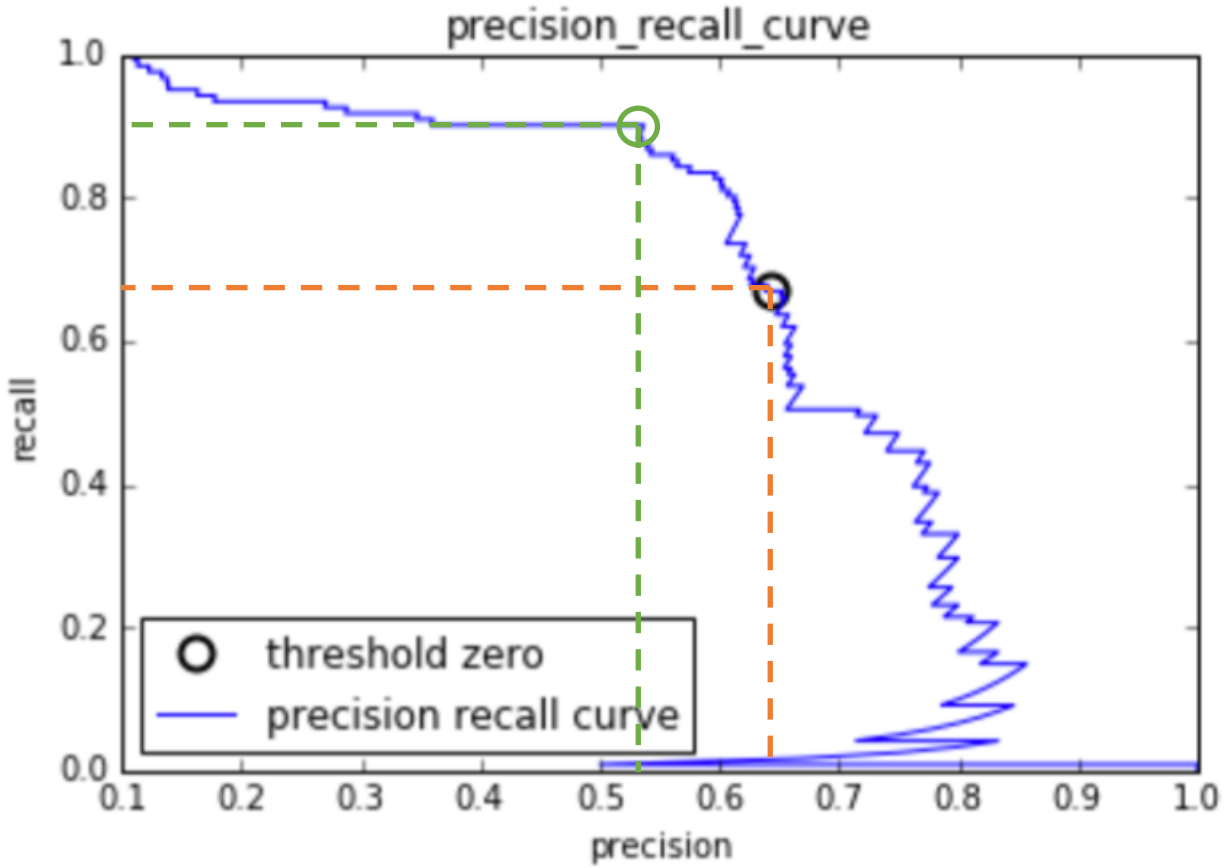


Raising the threshold moves the curve towards *higher precision* and *lower recall*

The more our model keeps *recall* high while increasing *precision*, the better

HPSI
Human Performance Systems, Inc.

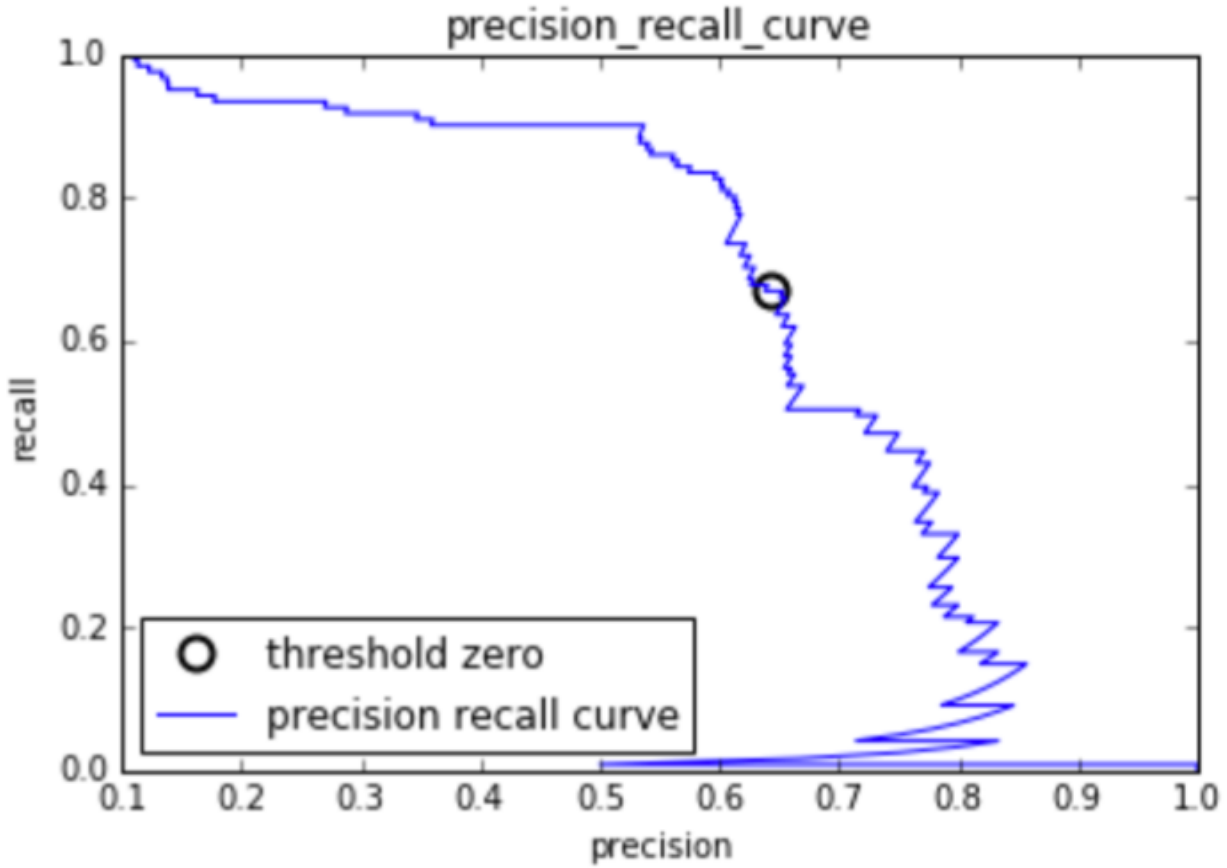
Precision-Recall Curves



Looking at this curve, it is possible to achieve a much higher recall, while keeping precision above 0.5, by decreasing the threshold

In the cancer example, we would consider moving the threshold to the one in the green circle

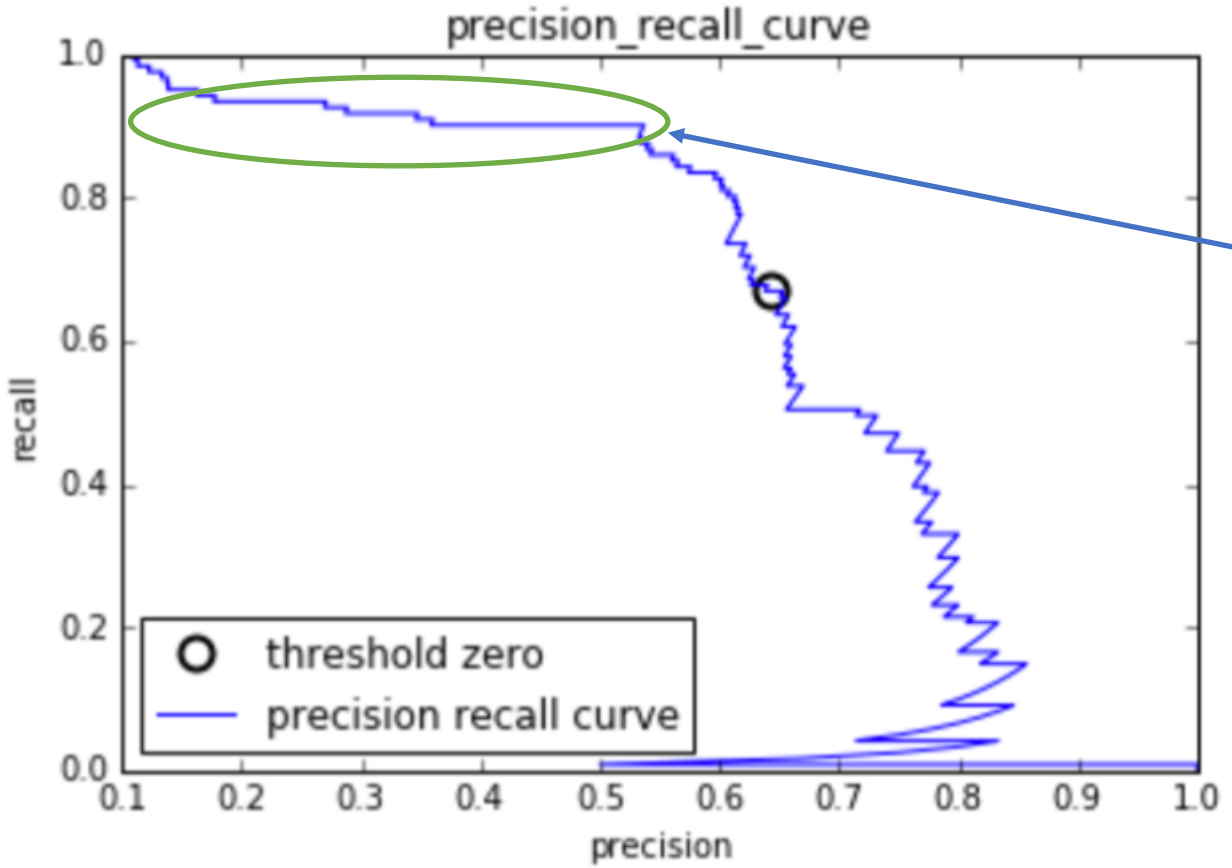
Precision-Recall Curves



Conversely, in this example, getting a high precision is very costly

HPSI
Human Performance Systems, Inc.

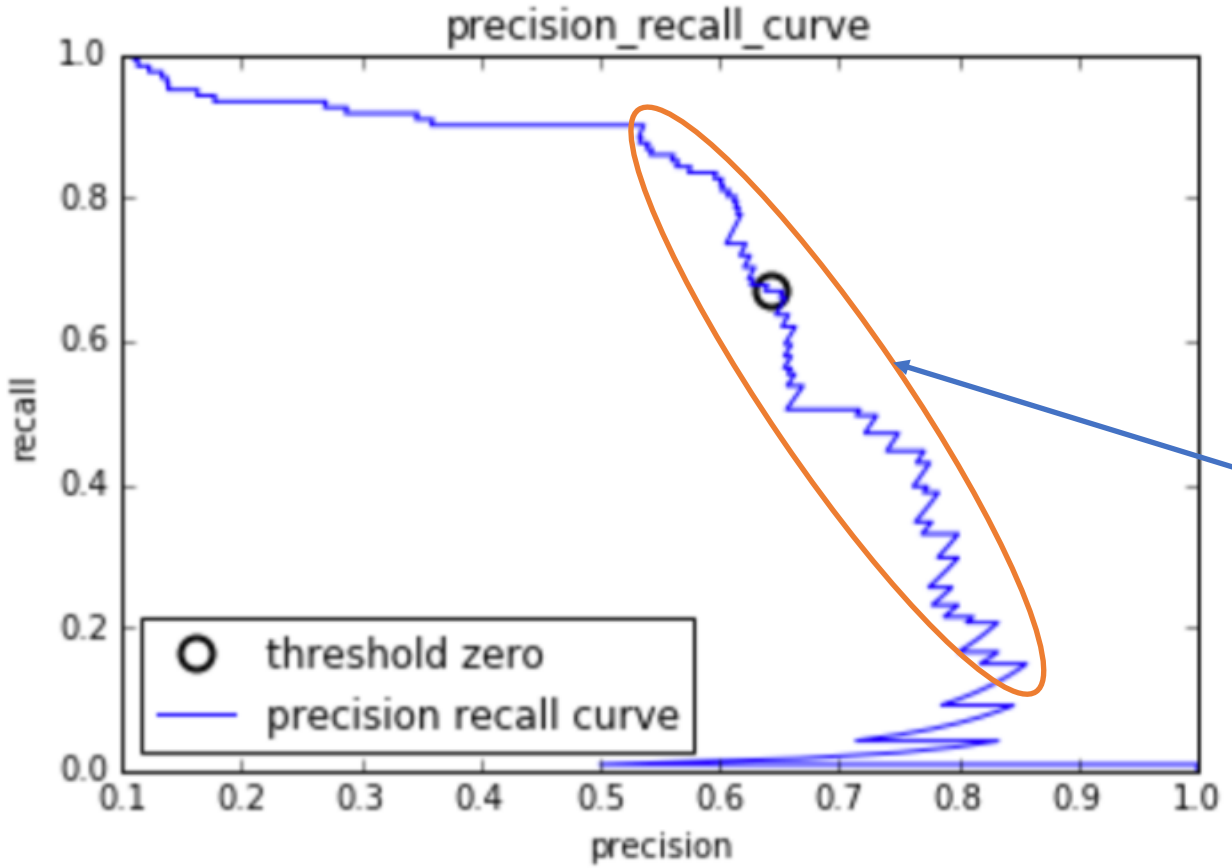
Precision-Recall Curves



Conversely, in this example, getting a high precision is very costly

This flat region allows us to increase precision at very little cost to recall

Precision-Recall Curves



Conversely, in this example, getting a high precision is very costly

But, for each increase in precision past 0.5, it costs us a lot of recall (evidenced by the steep slope)

ROC Curve

- Another common tool to evaluate a model at different thresholds is the *Receiver Operating Characteristics (ROC) Curve*
- The ROC curve similarly considers all possible thresholds for a given classifier
  - But, instead of reporting precision and recall, it shows the *false positive rate* (FPR) against the *true positive rate* (TPR)
  - Note: the true positive rate is simply another name for *recall*, while the false positive rate is the fraction of false positives out of all negative samples

ROC Curve in sklearn

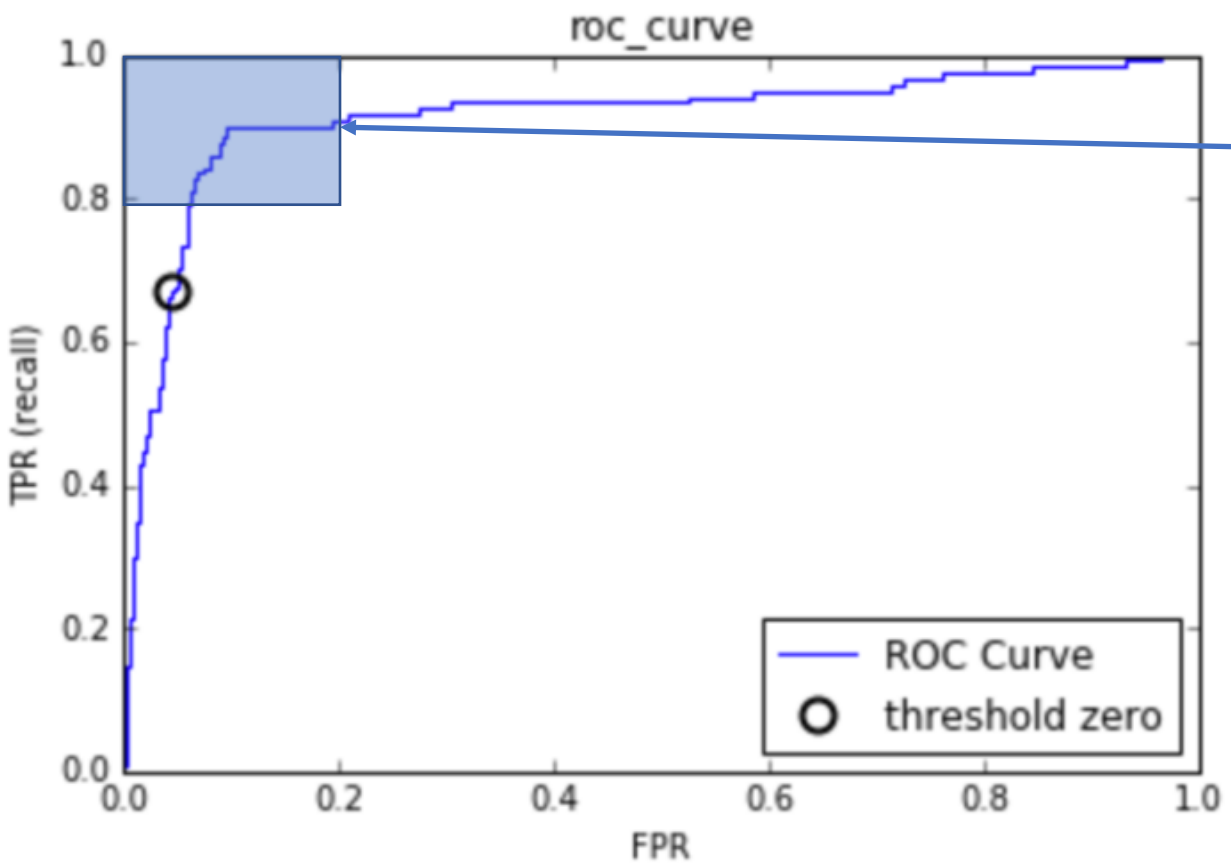
```
>>> from sklearn.metrics import roc_curve
>>> fpr, tpr, thresholds = roc_curve(y_test,
model.decision_function(X_test))
```

- The `roc_curve` function returns a list of false positive rates and true positive rates for all possible thresholds (all values that appear in the decision function) in sorted order
- We can use this to plot the curve

ROC Curve in sklearn

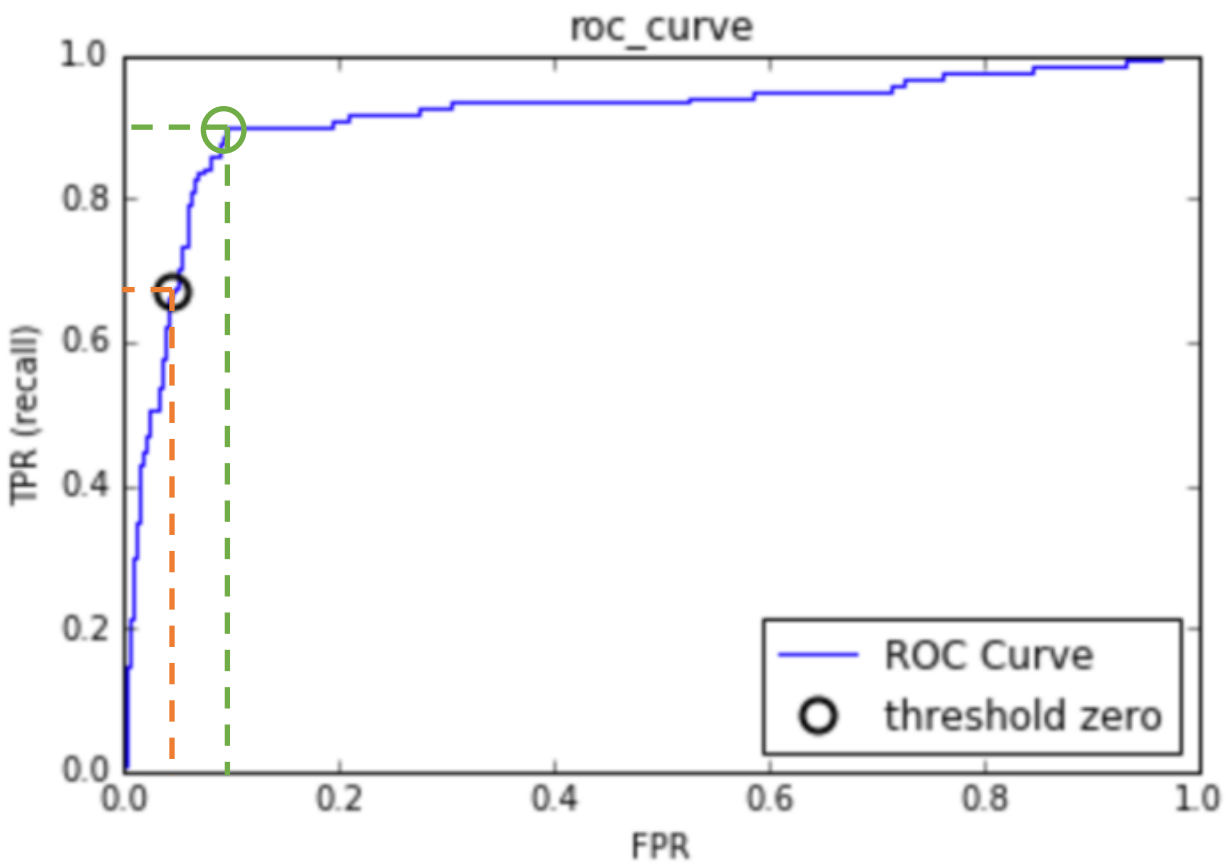
```
# find threshold closest to zero:close_zero =
np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o',
markersize=10, label="threshold zero",
fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
plt.title("roc_curve")
```

ROC Curve



For the ROC curve, the ideal curve is close to the top left—you want a classifier that produces a *high recall* while keeping a *low false positive rate*.

ROC Curve



Compared to the default threshold of zero, the curve shows that we could achieve a significantly higher recall (around 0.9) while only increasing the FPR slightly

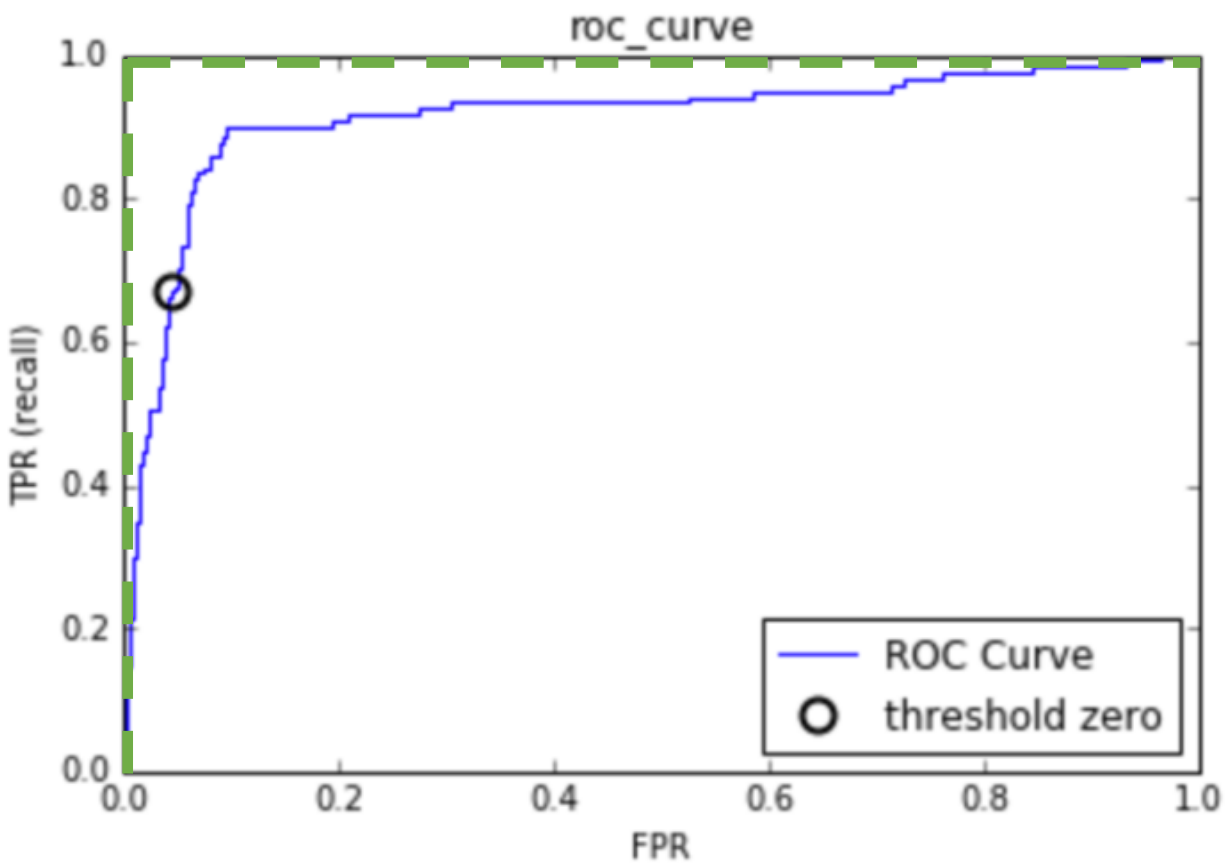
AUC

- If we want to summarize the ROC Curve, we can compute the *area under the curve (AUC)*
  - Bounded between [0, 1]
  - Predicting randomly always produces an AUC of 0.5, not matter how imbalanced the classes in a dataset are
  - This makes it a much better metric for imbalanced classification problems than accuracy
  - The AUC can be interpreted as evaluating the *ranking* of positive samples
  - If you randomly chose one positive and one negative observation, AUC represents the likelihood that your classifier will assign a **higher predicted probability** to the positive observation
  - An AUC of 1 means that all positive points have a higher score than all negative points

AUC in sklearn

```
>>> from sklearn.metrics import roc_auc_score
# use predicted probability for positive class
>>> auc = roc_auc_score(y_test,
model.predict_proba(X_test)[:, 1])
```

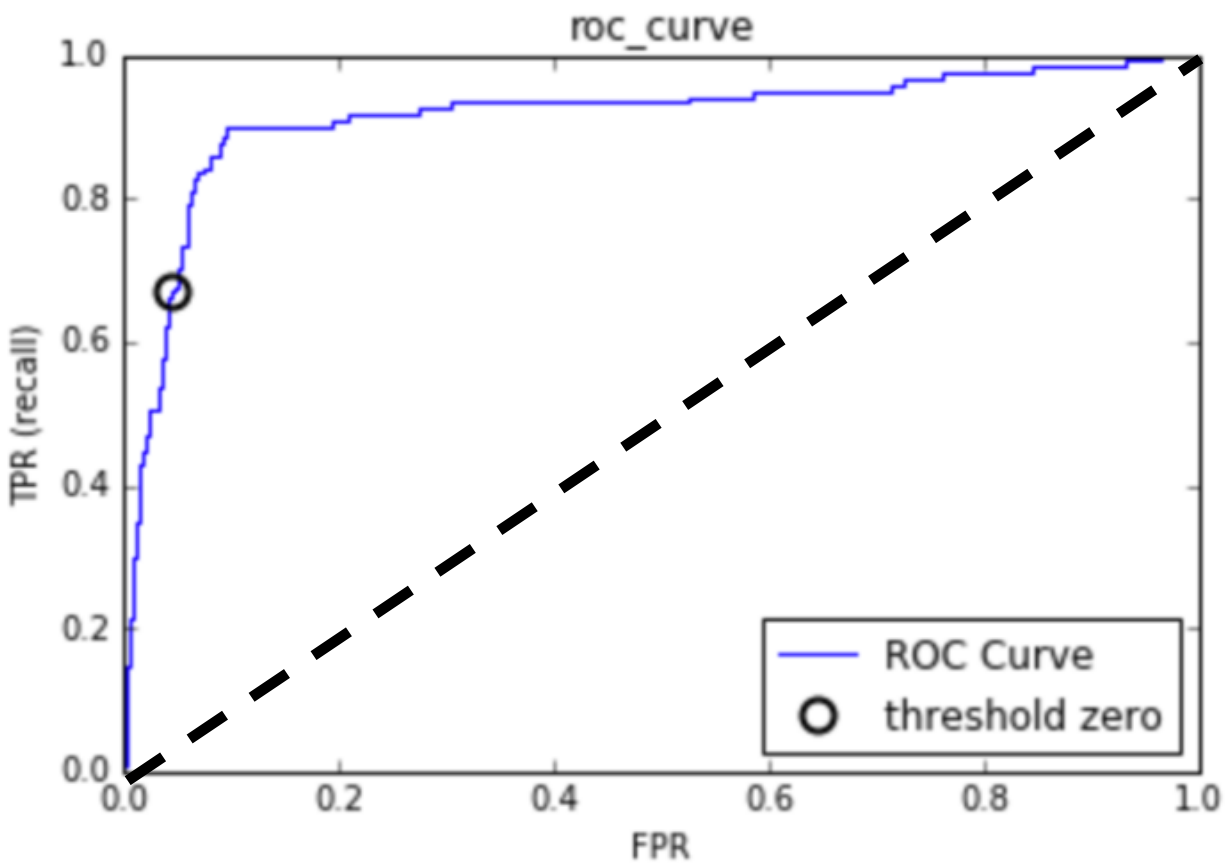
ROC Curve & AUC



A perfect ROC curve is shown in green

HPSI
Human Performance Systems, Inc.

ROC Curve & AUC

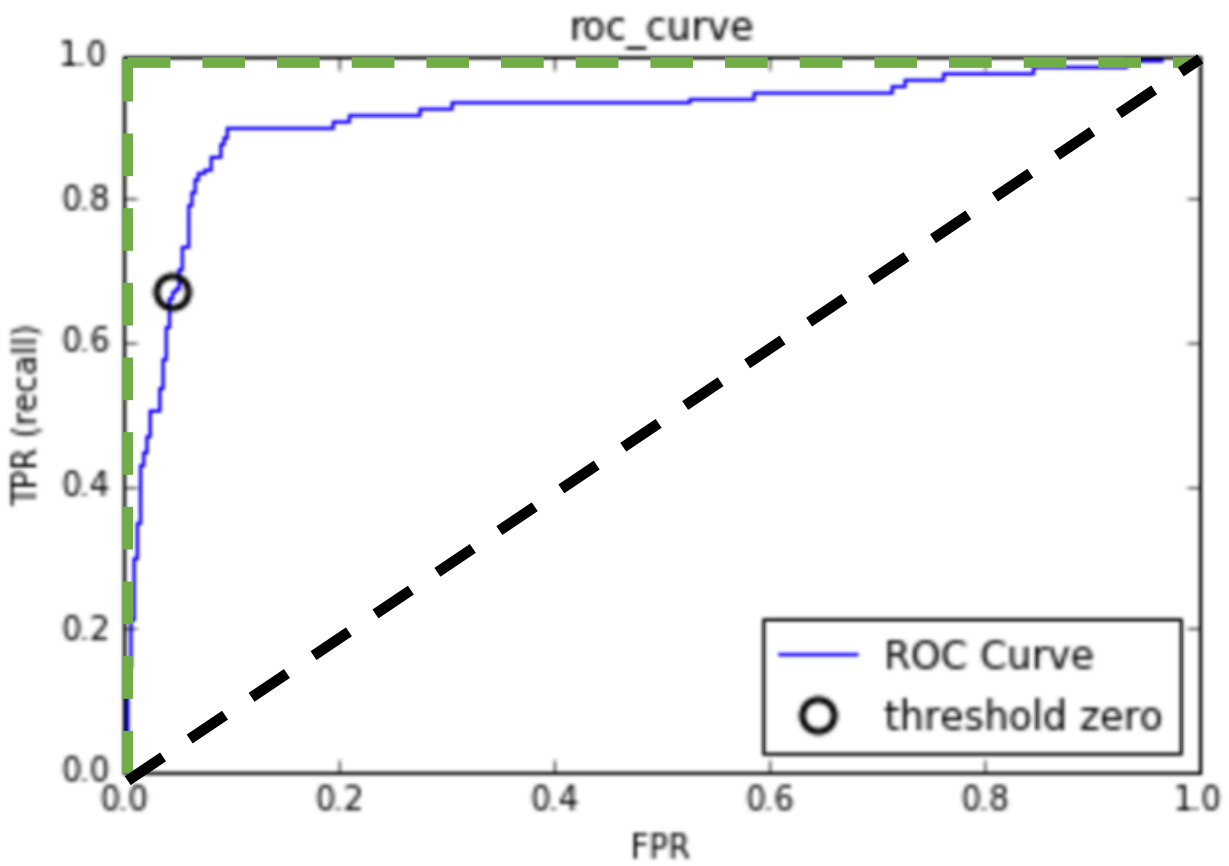


A perfect ROC curve is shown in green

If we draw a diagonal line on the ROC plot, this represents a model in which we guess randomly

HPSI
Human Performance Systems, Inc.

ROC Curve & AUC

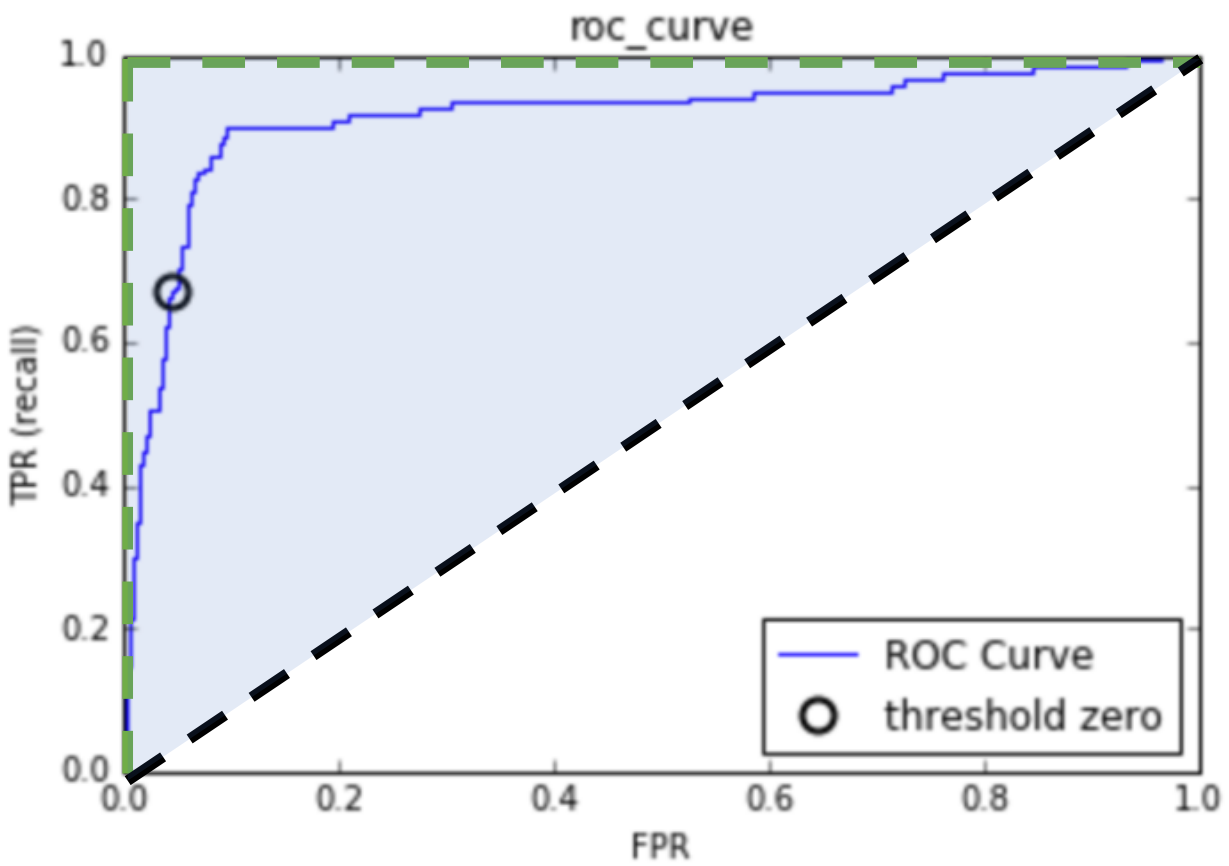


A perfect ROC curve is shown in green

If we draw a diagonal line on the ROC plot, this represents a model in which we guess randomly

Predicting randomly always produces an AUC of 0.5, no matter how imbalanced the classes in a dataset are

ROC Curve & AUC



A perfect ROC curve is shown in green

If we draw a diagonal line on the ROC plot, this represents a model in which we guess randomly

Predicting randomly always produces an AUC of 0.5, no matter how imbalanced the classes in a dataset are

Our model should lie somewhere between the ideal curve and the diagonal line

HPSI
Human Performance Systems, Inc.

Final Thoughts on ROC Curve & AUC

- Keep in mind that AUC is just a calculation of the area under the ROC curve
- It does not take the default threshold into account
- We still need to look at the ROC Curve

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

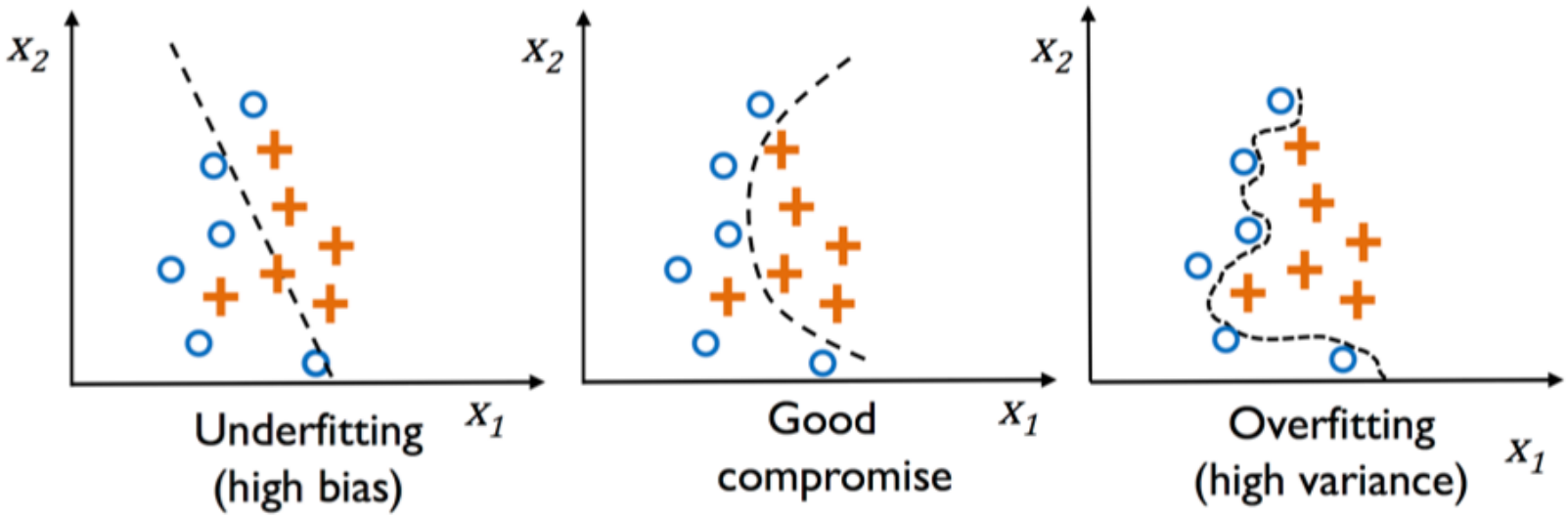
Model Validation

- Recall: one of the main goals in building our classification model is for it perform well on unseen data (*generalize*)
- For this reason, we split our data up into a training set and a test set
- But, when we evaluate the performance of our model, we may find it performs much better on the training set than the test set
- In this section, we will talk more about how to make our models generalize better and more robust methods for evaluating generalization performance than a simple split of the original data

Generalization, Overfitting, & Underfitting

- If we notice our model performs much better on the training set than the test set, this is a strong indication of *overfitting*
- ***Overfitting*** occurs when our model is too complex given the underlying training data
  - In this case, we say the model has ***high variance***
  - *Learning the random noise in the data, not the patterns*
- Conversely, ***Underfitting*** occurs when our model is not complex enough to capture the patterns in the training data well and also suffers from low performance on unseen data
  - In this case, we say the model has ***high bias***

Generalization, Overfitting, & Underfitting



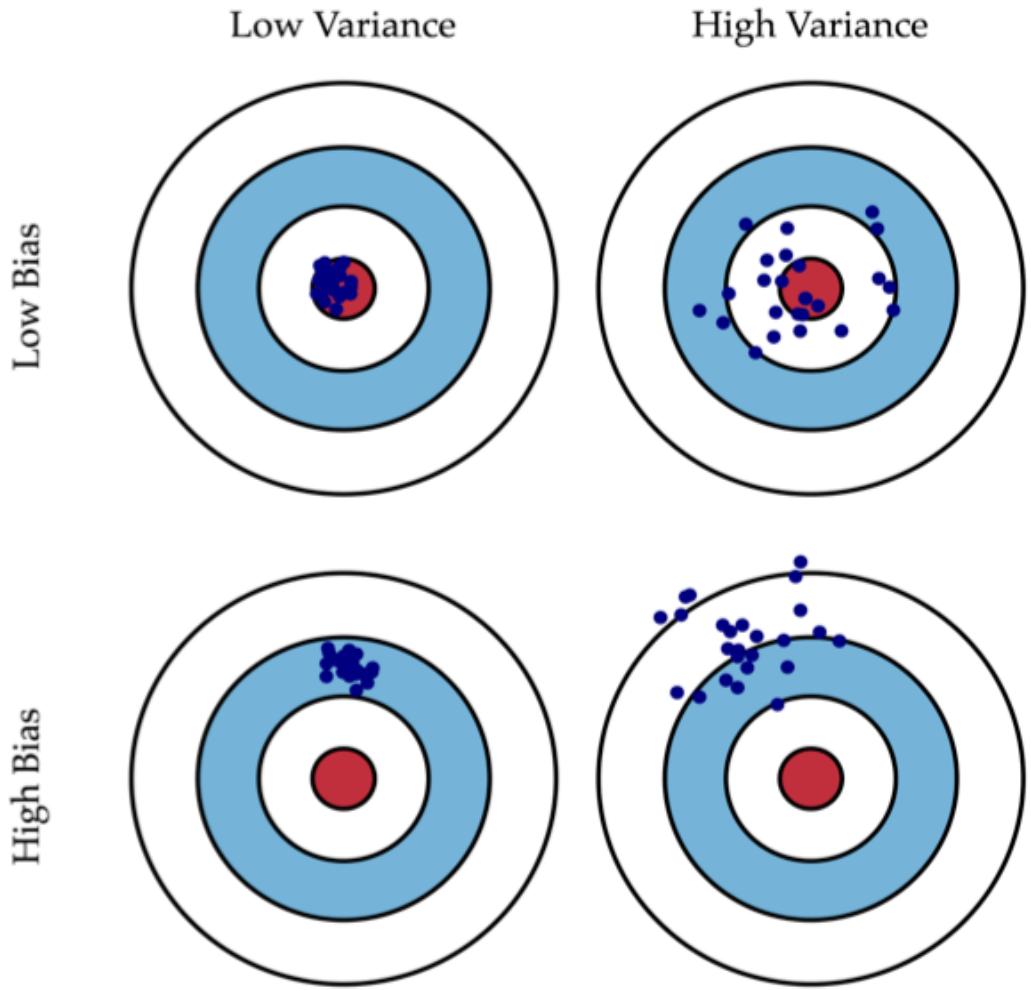
Generalization, Overfitting, & Underfitting

- So we want to avoid *overfitting* because it gives too much predictive power to noise in our training data.
- But in our attempt to reduce overfitting we can also begin to *underfit* or ignore important features in our training data
- So how do we balance the two?
  - This is a problem we call the ***Bias-Variance Tradeoff***

Bias-Variance Tradeoff

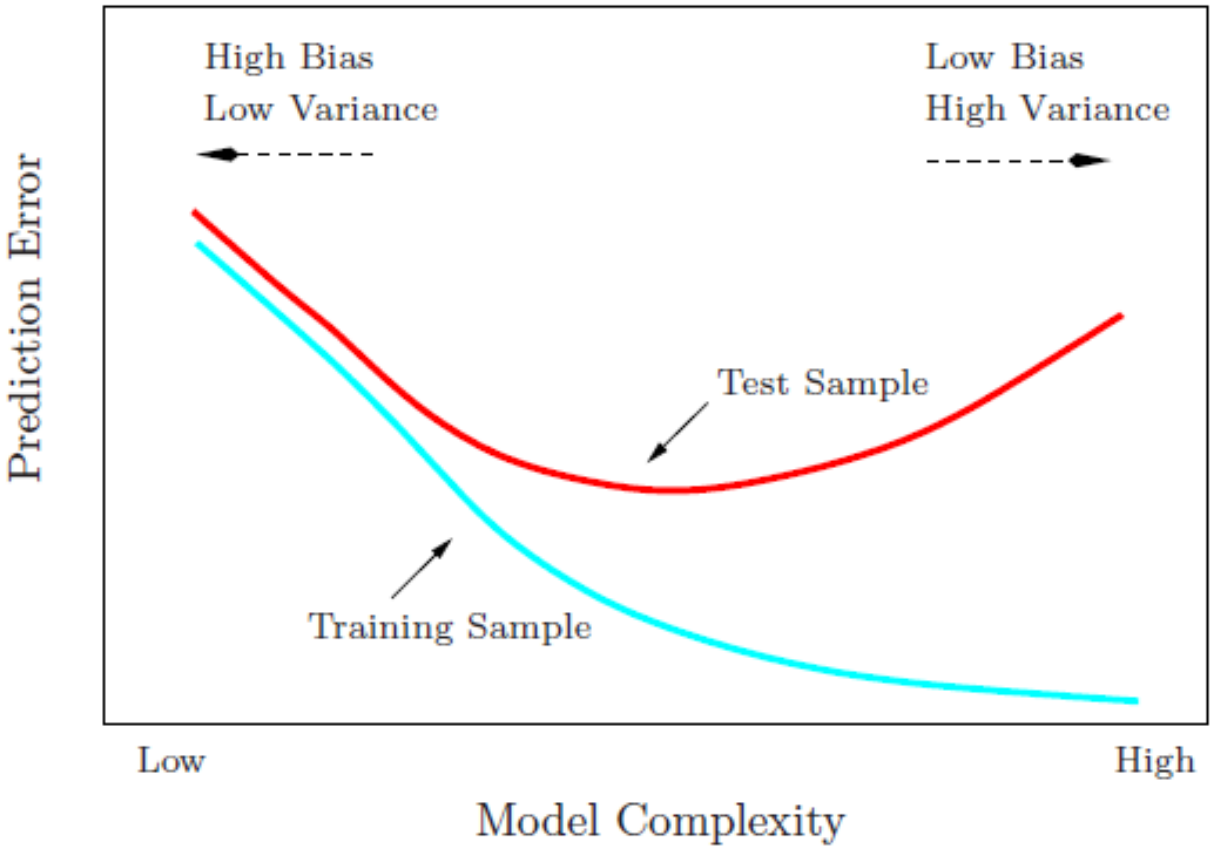
- ***Variance*** measures the consistency (or variability) of the model prediction for a particular sample instance if we were to retrain the model multiple times on different subsets of the training dataset
  - The model is sensitive to the randomness in the training data
- ***Bias*** measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets
  - Bias is the measure of the systematic error that is not due to randomness

Bias-Variance Tradeoff

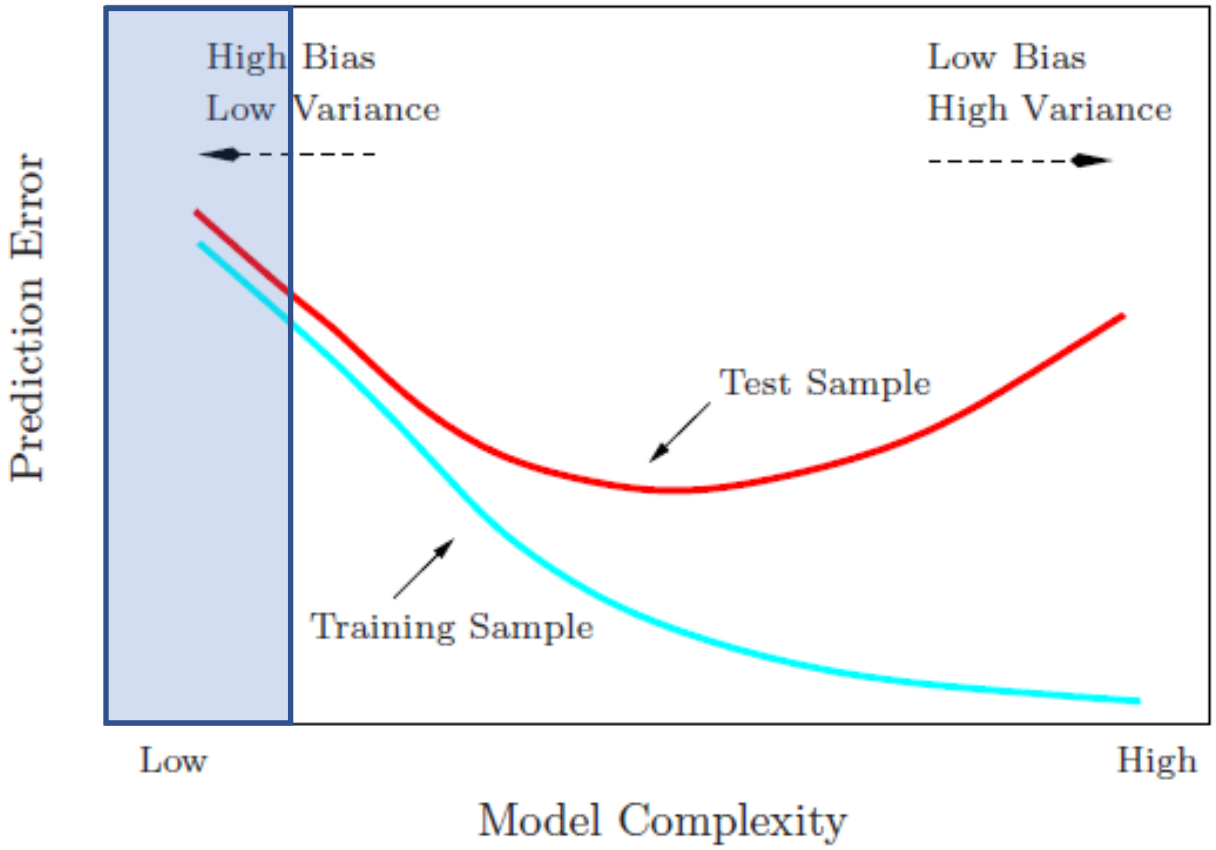


HPSI
Human Performance Systems, Inc.

Bias-Variance Tradeoff

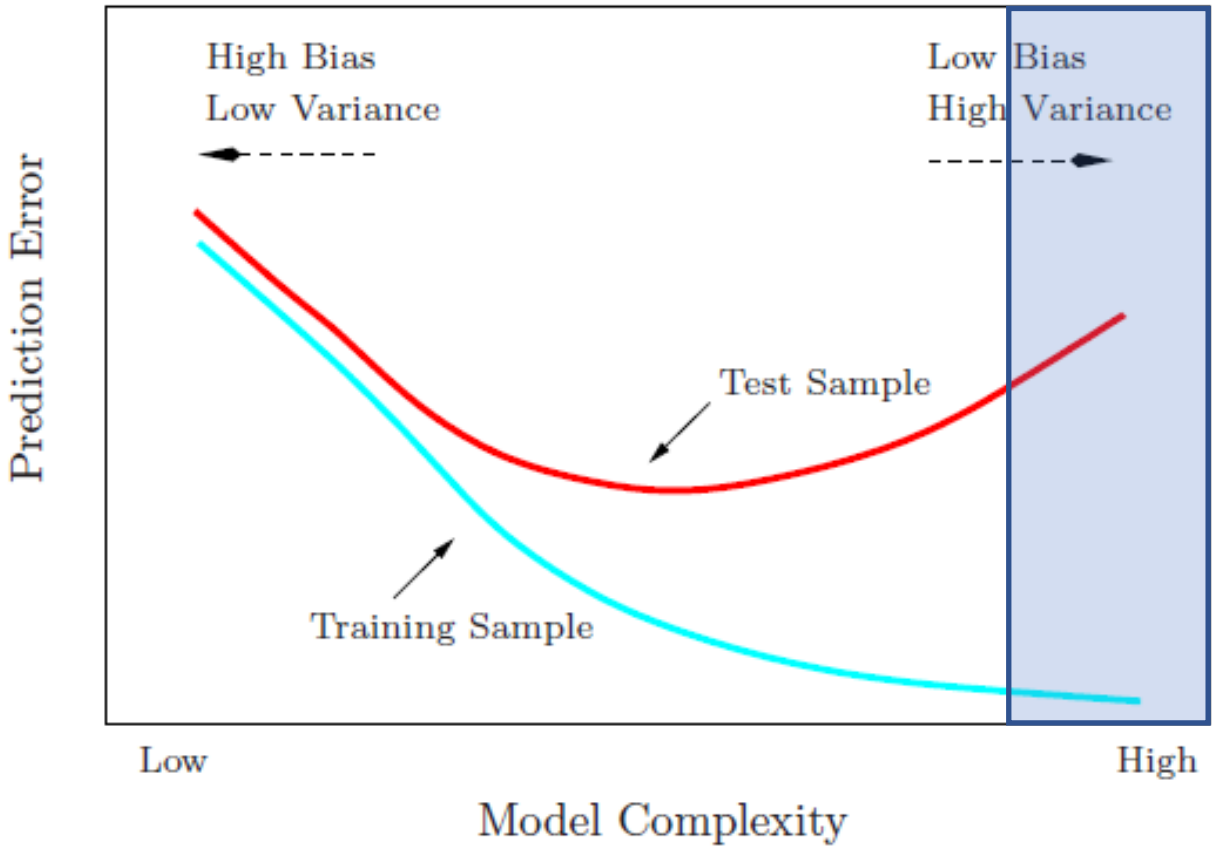


Bias-Variance Tradeoff



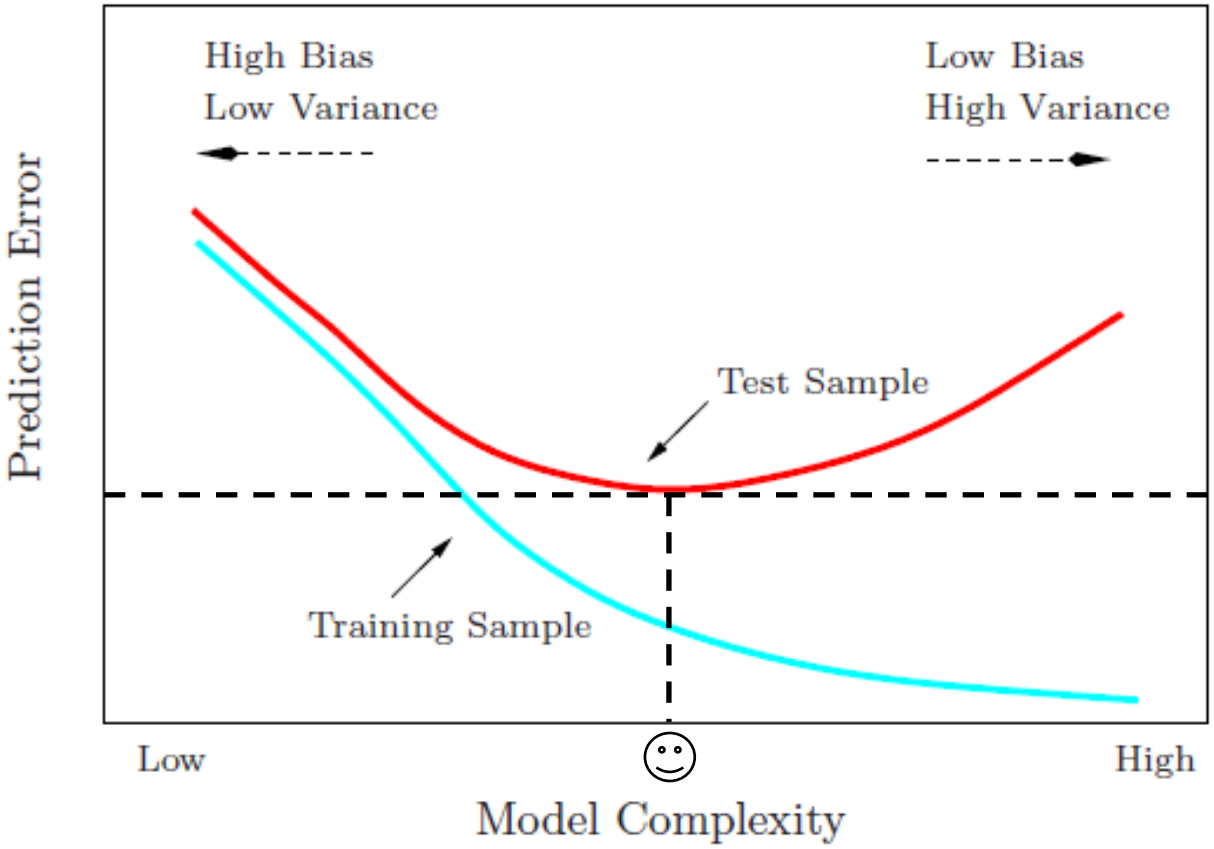
Low complexity models will result in high error (poor accuracy) for both training and test data. This is because the model lacks enough complexity to describe the data.

Bias-Variance Tradeoff



High complexity models will result in a low training error and a high test error. This is because a complex model will be able to model the training data a bit *too* well, and can't generalize to the test data.

Bias-Variance Tradeoff



The ***best complexity*** lies where the *test error* reaches a *minimum.*

Bias-Variance Tradeoff

- As we've seen, overfitting and underfitting have very clear signatures in training and test data
  - *Overfitting* results in *low training error* and *high test error*
  - *Underfitting* results in *high training & test errors*
- To find an acceptable bias-variance tradeoff, we need to evaluate our model carefully
- Often we use more robust methods than a simple training and test set split to find this tradeoff
  - ***Cross-validation***

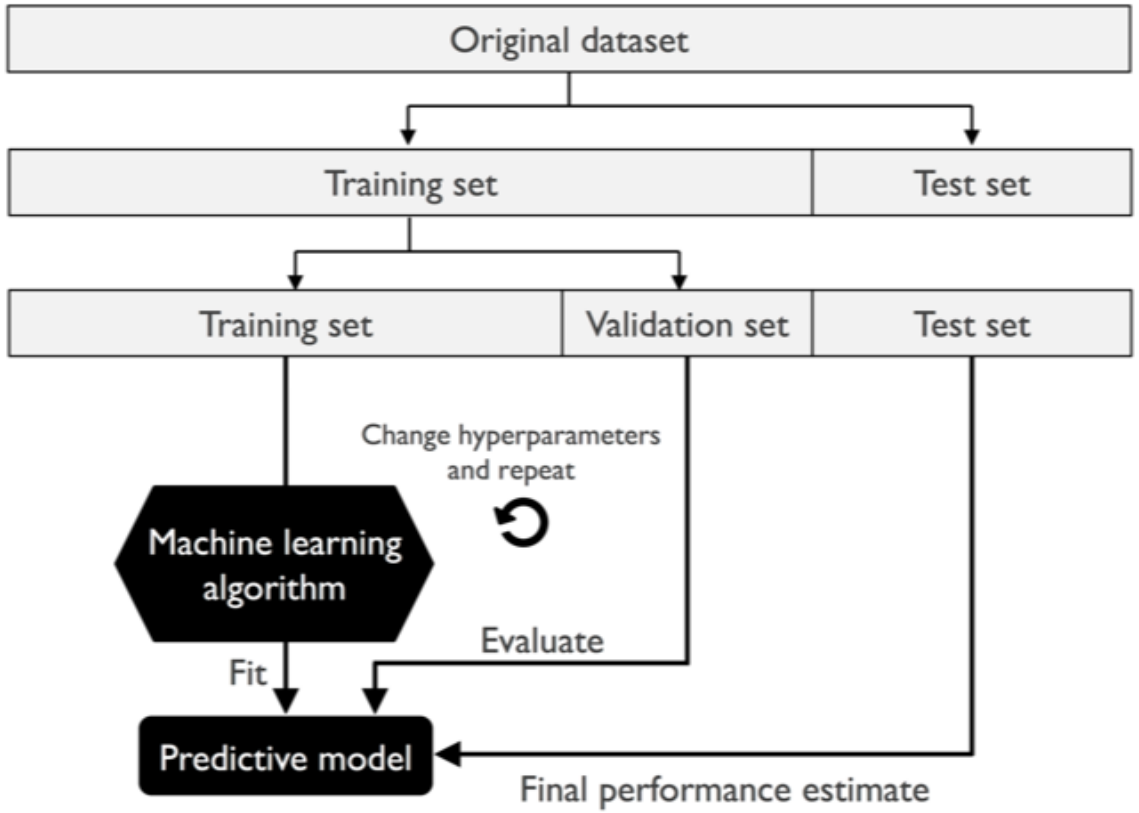
Cross-Validation

- In ***cross-validation***, instead of just splitting the data set in to a training set and a test set, the data is split repeatedly and multiple models are trained
- The *two* most common methods are:
  1. Holdout Cross-Validation
  2. *k*-Fold Cross-Validation

Cross-Validation

- Holdout Cross-Validation
  - Using the holdout method, we split our initial dataset into 3 separate sets:
    1. Training (70 %)
    2. Validation (15 %)
    3. Testing (15 %)
  - The training set is used to fit the models, and the performance on the validation set is then used for the model selection
    - ***Model Selection*** is the process where we try to select the optimal values of tuning parameters (*hyperparameters*) for our models
    - The advantage of having a test set that the model hasn't seen before during the training and model selection steps is that we can obtain a less biased estimate of its ability to generalize to new data

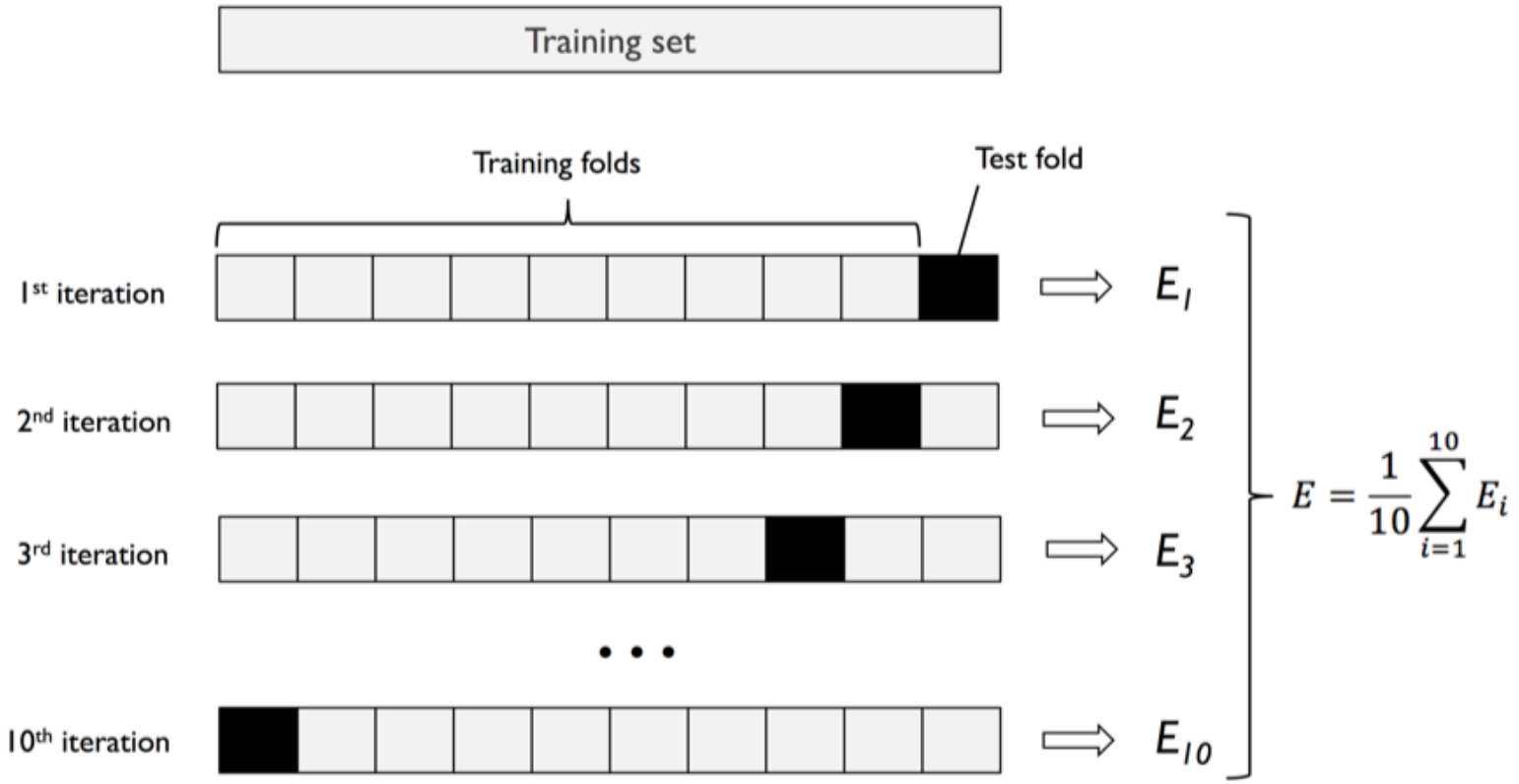
Holdout Cross-Validation



Cross-Validation

- *k*-Fold Cross-Validation
  - In *k*-fold cross-validation, we randomly split the training dataset into *k* folds without replacement, where *k-1* folds are used for the model training, and one fold is used for performance evaluation
  - This procedure is repeated k times so that we obtain k models and performance estimates

k-Fold Cross-Validation



HPSI
Human Performance Systems, Inc.

k-Fold Cross-Validation

- We then calculate the average performance of the models based on the different, independent folds to obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method
- Generally use *k*-fold cross-validation for model tuning
- Once we have found satisfactory *hyperparameter* values, we can retrain the model on the complete training set and obtain a final performance estimate using the independent test set

Standard ML workflow for classification:

1. **Define** the problem
2. **Collect** Data (with target labels)
3. **Explore** the Data (EDA)
4. **Preprocess** the Data
5. **Fit** Model on Training Set
6. **Predict** on Test Set
7. **Evaluate** Model Performance
8. **Validate** the Model
9. **Improve** Model Performance/**Select** the Best Model

Improve Model Performance/Select the Best Model

- Gather more training samples
  - Not always possible
- Merge additional datasets
  - Not always possible
- Tune hyperparameters
  - This is model-specific and will be covered in the following days
- Adjust model complexity & flexibility (bias-variance tradeoff)

Machine Learning Overview Wrap-up

- Let's do some more examples and exercises to reinforce the concepts from the machine learning overview!