Data Foundations: Feature Extraction and Missing Data

Instructor: Anthony Rios

Outline

Scikit-learn Pipelines

A **Pipeline** is sequence of transformations (find a set of features, generate new features, etc...) before applying a final estimator/classifier.

For text classification, before applying a classifier, we must first transform the text into a feature matrix.

There are many options to use when using CountVectorizer: how many ngrams? Should we remove stopwords? How about remove works that occur less than 5 times?

All operations can be combined into a single object using "Pipelines".

Scikit-learn Pipelines

```
>>> from sklearn.pipeline import Pipeline

pipeline = Pipeline([
        ('vect', CountVectorizer()),
        ('tfidf', TfidfTransformer()),
        ('clf', LinearSVC())]
```

Items in **red** represent component names. **blue** represents instances of sklearn objects.

Here we assume we already imported CountVectorizer, TfidfTransformer, and LinearSVC

Pipelines

Given the pipeline:

```
pipeline = Pipeline([
        ('vect', CountVectorizer()),
        ('tfidf', TfidfTransformer()),
        ('clf', LinearSVC())])
```

we can train a new classifier directly starting with the text.

```
>>> train_txt = ['test text 1', 'test text 2', 'test text 3']

>>> y = ['class1', 'class2', 'class1']

# Trains a classifier using the entire pipeline
>>> pipeline.fit(train_txt, y)
```

GridSearch with Pipelines

```
>>> from sklearn.model_selection import GridSearchCV

pipeline = Pipeline([
        ('vect', CountVectorizer()),
        ('tfidf', TfidfTransformer()),
        ('clf', LinearSVC())])

>>> params = {'vect__ngram_range':((1,1), (1,2))
              'clf__C': (0.1, 1.)}
>>> train_txt = ['test text 1', 'test text 2', 'test text 3']

>>> y = ['class1', 'class2', 'class1']

>>> clf = GridSearchCV(pipeline, params, cv=3)

>>> clf.fit(train_txt, y) # Trains pipeline with GridSearchCV
```

Exercise 1

Expand on the previous class's in-class exercise to include CountVectorizer parameters in GridSearchCV with Pipelines. Include the following CountVectorizer Parameters: stop_words = ['english', None], lowercase=[False, True], and min_df=[1, 5, 10]

What are the parameters that produce the best micro F1?

Feature Engineering

Feature **engineering** refers to the process of designing the types of features that will be used in a classifier or predictor

- In text, should we use the count of words? Counts of phrases? Weight the counts some how?

- In images, should we use the raw pixels? Counts of colors? Or something else?

There are standard types of features that are commonly used for various data types, but sometimes you need to design your own. **BE CREATIVE!!**

Dimensionality

The **dimensionality** of data is the number of variables

- Usually this refers to the number of input variables

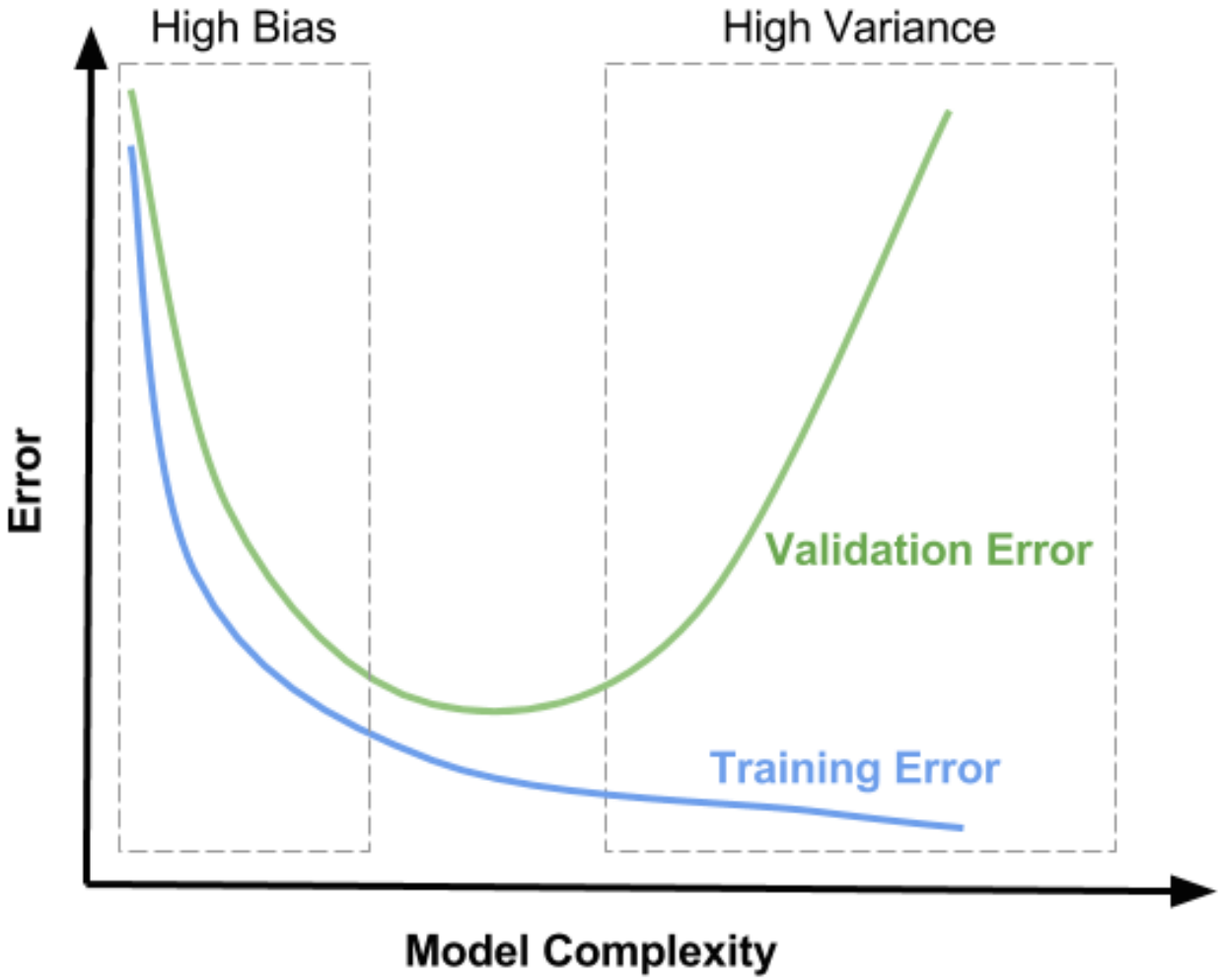
- In other words, the number of **features**

The **curse of dimensionality** refers to challenges that arise when data has many dimensions

- Training: the more features you have, the more data you need

- Distance: all points are far apart in high-dimensional space, hard to define "close" vs "far". More specifically, the distances are uniformly distributed across all pairs of data points.

Bias Variance Trade Off



Dimensionality Reduction

Dimensionality reduction refers to the process of reducing the number of features in your data

- One method is **feature selection** which reduces the number of features, but keeps the original feature values.

- There is also **feature extraction/transformation** that **transforms** the feature space, creating features that are different from the original feature types.

Feature Selection

Once you've generated all features you want, you may want to select only a subset of them

Why?

- To many features may result in overfitting.
  - Regularization can also address this
- Too many features make training and prediction slower
  - Efficiency is a big reason to do feature selection, especially if real-time processing is needed

Feature Selection

How can we pick features?

- Extrinsic 'wrapper' approaches:
  - For each subset of features, build and evaluate a classifier for some task.
  - Pick subset of features with the best performance
- Intrinsic 'filtering' methods:
  - Use some intrinsic (statistical?) measure
  - Pick features with highest scores

Feature Selection

- Wrapper approach:
  - Easy to understand, implement (We did this with grid search)
  - Clear relationship between selected features and task performance
- Cons:
  - Computationally intractable – $2^{\#\text{ features}}$ combinations
  - Specific to task/classifier
- Filtering approach:
  - Pros: theoretical basis, less task/classifier specific
  - Cons: Does not always boost task performance

Feature Selection by filtering

Main Idea: rank features according to a predetermined numerical function that measures the "**importance**" of the terms

It is fast and classifier-independent

Scoring functions:

- Information Gain
- Mutual Information
- Chi Square

Feature Selection Methods

Question: What makes a good feature?

Intuition: for a category c_i, the most valuable feature are those that are distributed most **differently** in the sets of **positive** and **negative** examples of c_i

Feature Selection Methods: DF

Document Frequency (DF):

- Number of documents in which word t_k appears

Applying DF:

- Remove terms with DF below some threshold

Intuition:

- Vary rare terms will **not** help with categorization.

Pros: Easy to implement, scalable

Cons: Ad-hoc, low DF terms 'topical'

Basic Notation/Distributions

- Assume binary representation of terms/classes
- t_k: term/word in T; c_i: class in C
- $P(t_k)$: proportion of documents in which t_k appears
- $P(c_i)$: proportion of documents of class c_i
  - Binary so have

    $P(\bar{t}_k)$, $P(\bar{c}_i)$

    $P(t_k, c_i)$, $P(\bar{t}_k, c_i)$, etc...

Calculating Basic Distributions

	$\bar{c}_i$	c_i
$\bar{t}_k$	a	b
t_k	c	d

$N = a + b + c + d$

$P(t_k, c_i) = d/N$ **Prob. of t_k and c_i**

$P(t_k) = (c + d)/N$ **Prob. of t_k**

$P(c_i) = (b + d)/N$ **Prob. of c_i**

$P(t_k|c_i) = d/(b + d)$ **Prob. of t_k given c_i**

Term Selection Methods: MI

Pointwise Mutual Information(MI)

$$MI(t_k, c_i) = \log(\frac{P(t_k, c_i)}{P(t_k)P(c_i)})$$

MI(t,c)=0 if t and c are independent

Issues: Can be heavily influenced by marginal probability

- Problems comparing terms of differing frequencies

Feature Selection with Scikit-Learn

```
>>> from sklearn.datasets import load_digits

>>> from sklearn.feature_selection import SelectKBest, chi2

>>> X, y = load_digits(return_X_y=True)

>>> X.shape
(1797, 64)

>>> skb = SelectKBest(chi2, k=20).fit(X, y)

>>> X_new = skb.transform(X)

>>> X_new.shape
(1797, 20)
```

Feature Selection with Scikit-Learn w/ Text

```
>>> from sklearn.feature_selection import SelectKBest, chi2

>>> X, y = #Assume is from CountVectorizer

>>> skb = SelectKBest(chi2, k=3).fit(X, y)

# Vec is from the CountVectorizer
>>> best =
np.array(vec.get_feature_names())[skb.get_support(indices=True)]

>>> print(best)
['word 1', 'word 2', 'word 3']
```

Exercise 2

Expanding on Exercise 1, and using the same data, do the following:

- Apply SelectKBest() with chi2 on the training data. What are the most important features with a k=10?
- Add SelectKBest() to the gridsearch **pipeline** from Exercise 1. Grid-Search over various choices of K. Choose K on your own.
  - Where should SelectKBest() be added to the pipeline?
  - What is the best K?
  - Did the performance improve over the model trained on the entire set of features?

Feature Transformation

In general:

- Original feature space has D dimensions (axes)

- New feature space has K dimensions (axes), $K < D$

The transformed feature vectors are sometimes called **embeddings**

Dimensionality Reduction

Why does dimensionality reduction work?

Why can it be automated?

Dimensionality Reduction

Intuition 1:

If multiple features are **correlated**, they can often be mapped to the same feature without losing a lot of information

Dimensionality Reduction

Intuition 2:

It's possible to change the dimensionallity so that instances that were similar to each other (or close to each other) in the original feature space are still similar to each other in the reduced space.

- and instances that were previously dissimilar should be dissimilar in the new space

You might imagine automating this, trying to learn a reduction that minimizes the different between the original similarities and new similarities

Dimensionality Reduction

Another intuition:

Think about dimensionality reduction as a **compression** problem (lossy compression)

- Want to compress the data as much as possible while retaining "most" of the information

- If you transform your feature vectors into new feature vectors with fewer dimensions, how well could you reconstruct the original vectors from the new (smaller) vectors?

Geometric Intuition

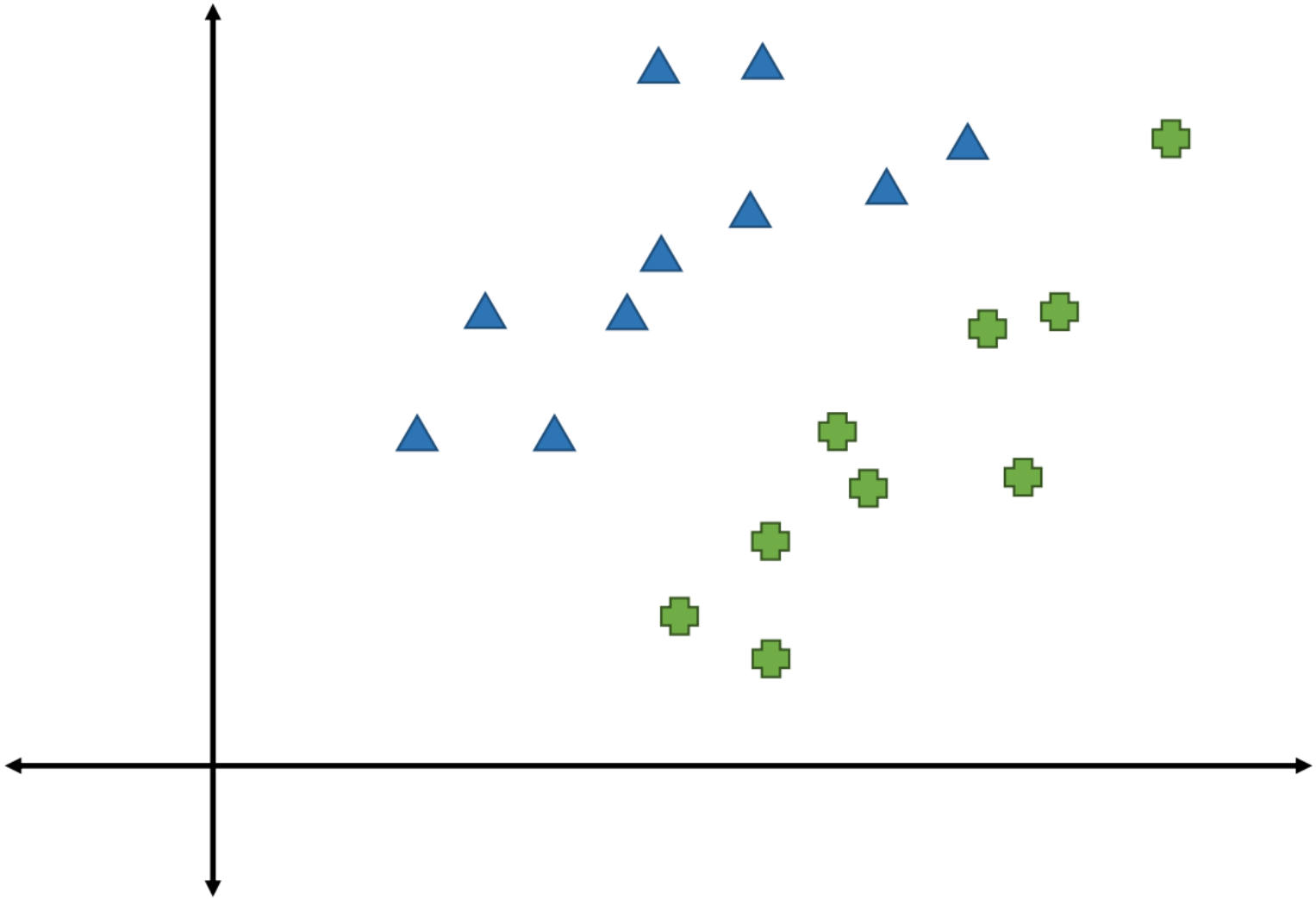
Like feature selection, transformation-based dimensionality reduction is usually automated

- You don't manually specify rules

In general, what does it mean to transform or change the features?

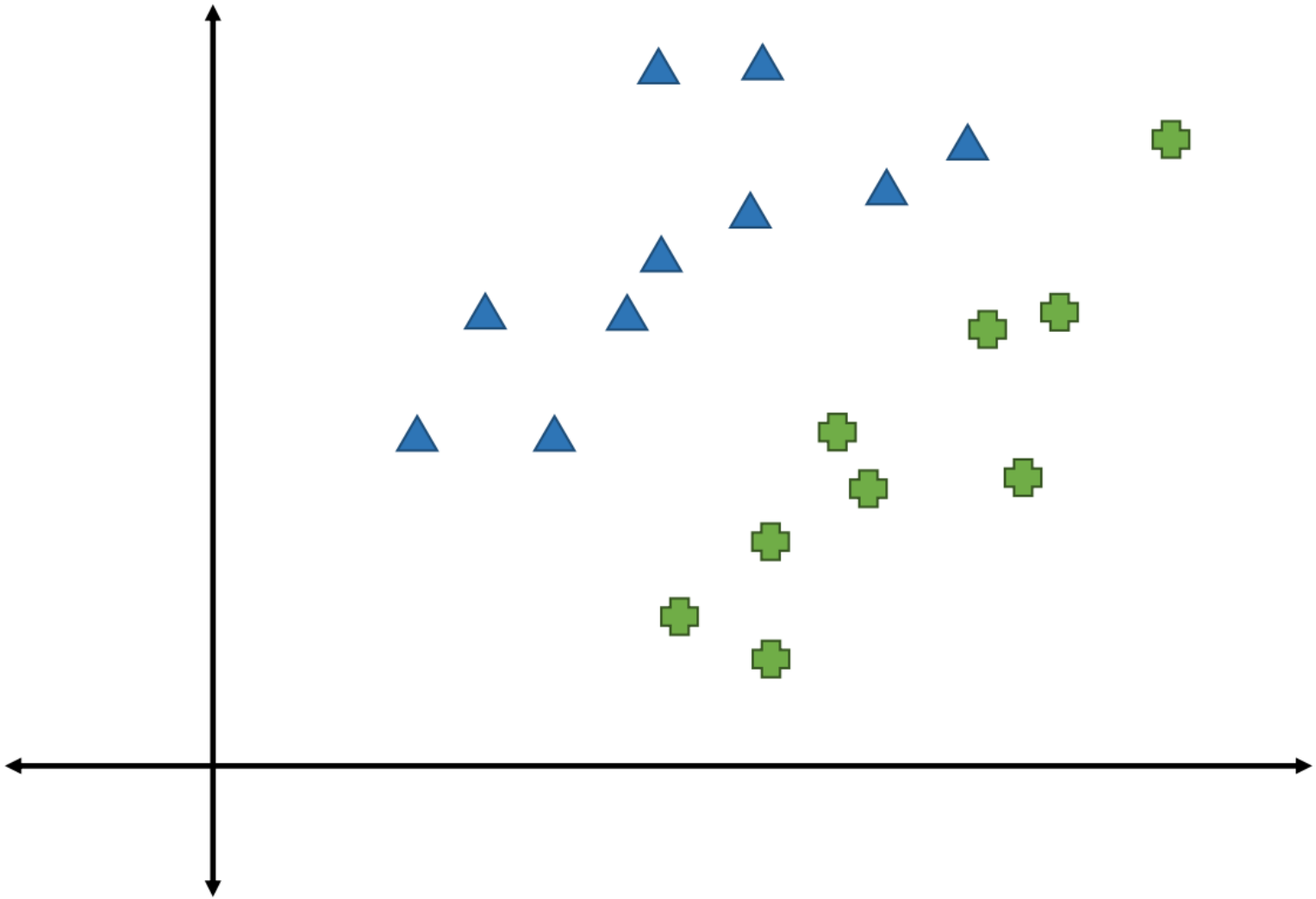
Geometric Intuition of Feature Selection and PCA

Suppose we have two dimensions (two features)



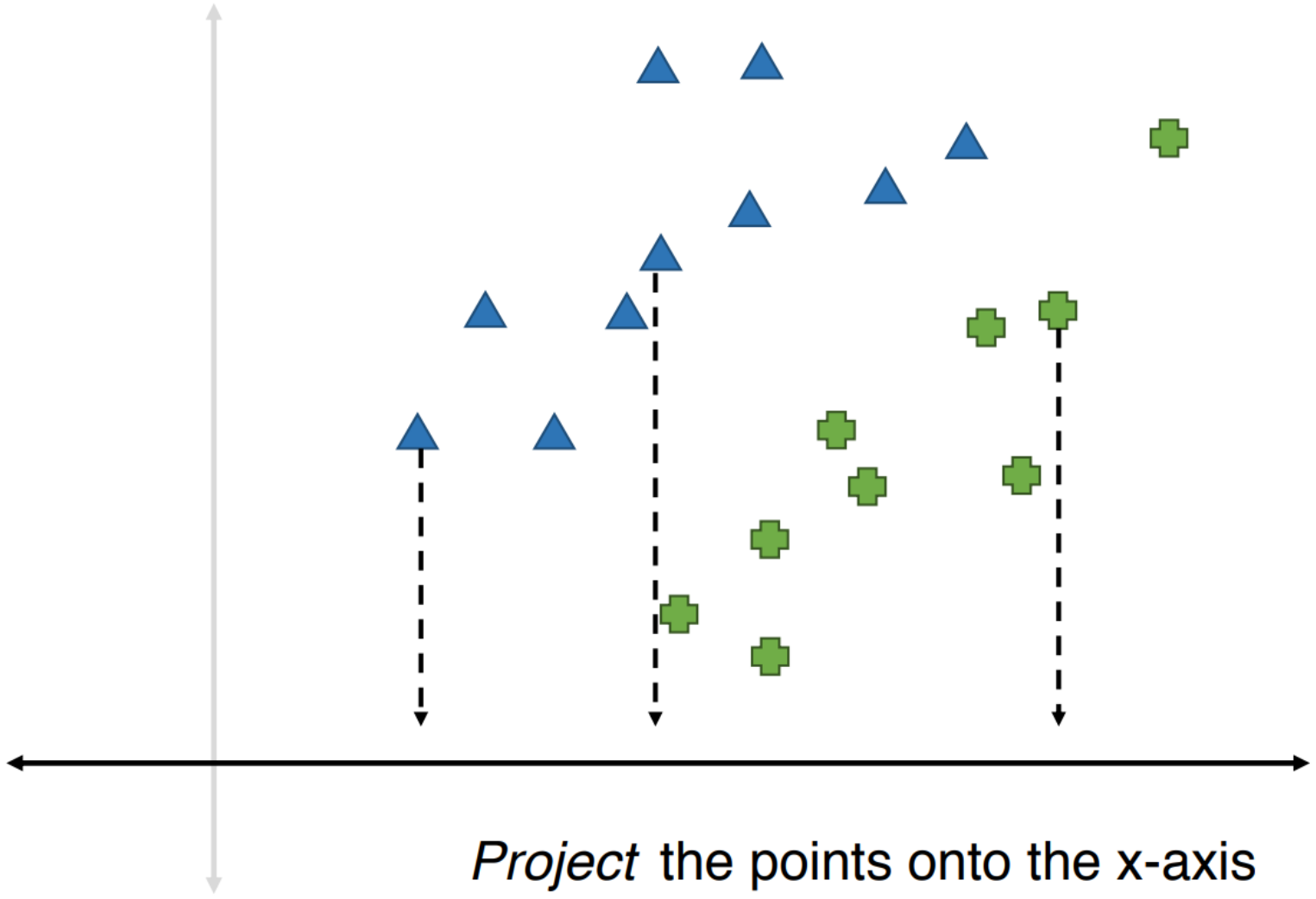
Geometric Intuition

Feature selection: choose one of the two features to keep



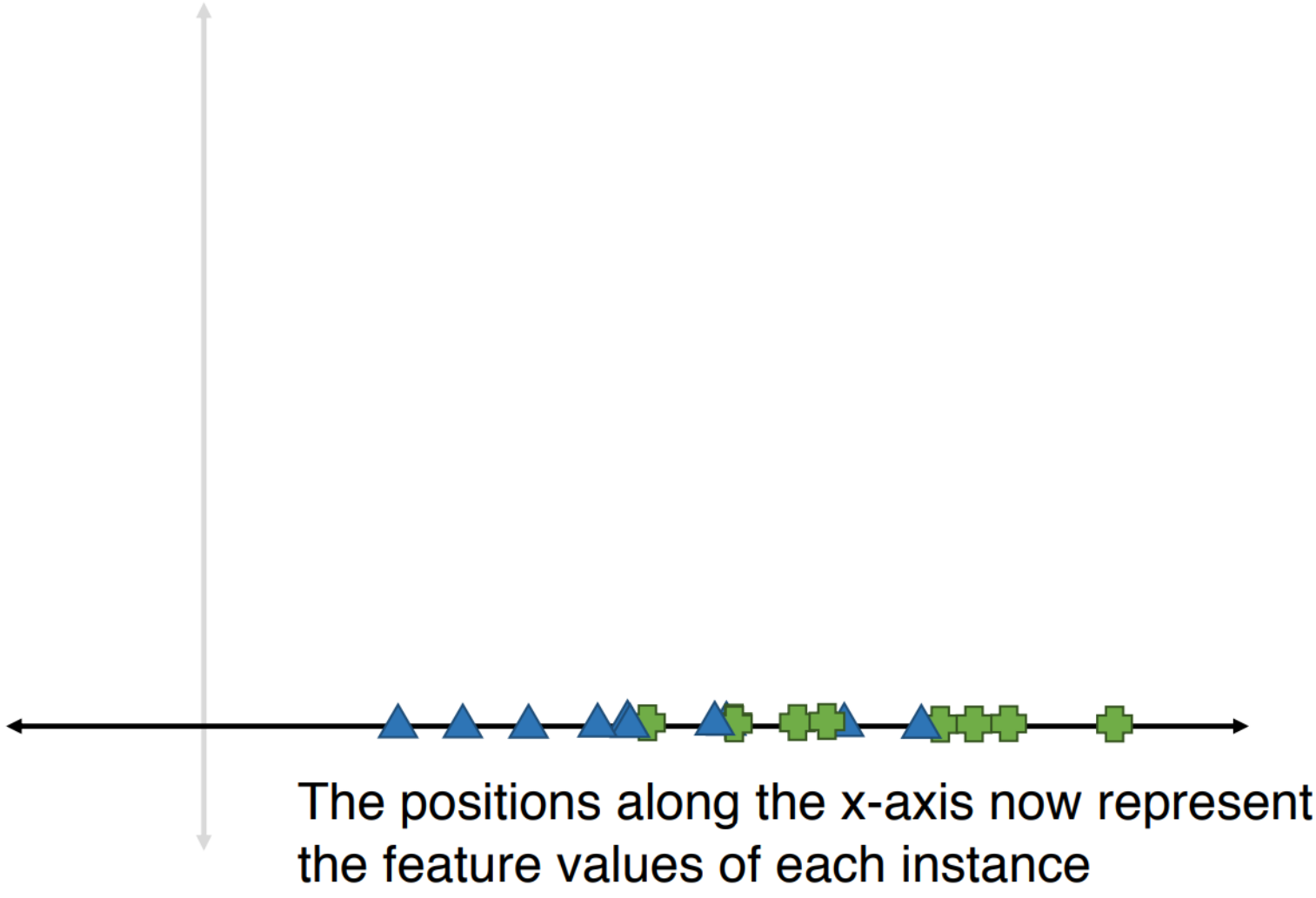
Geometric Intuition

Suppose we choose the feature represented by the x-axis



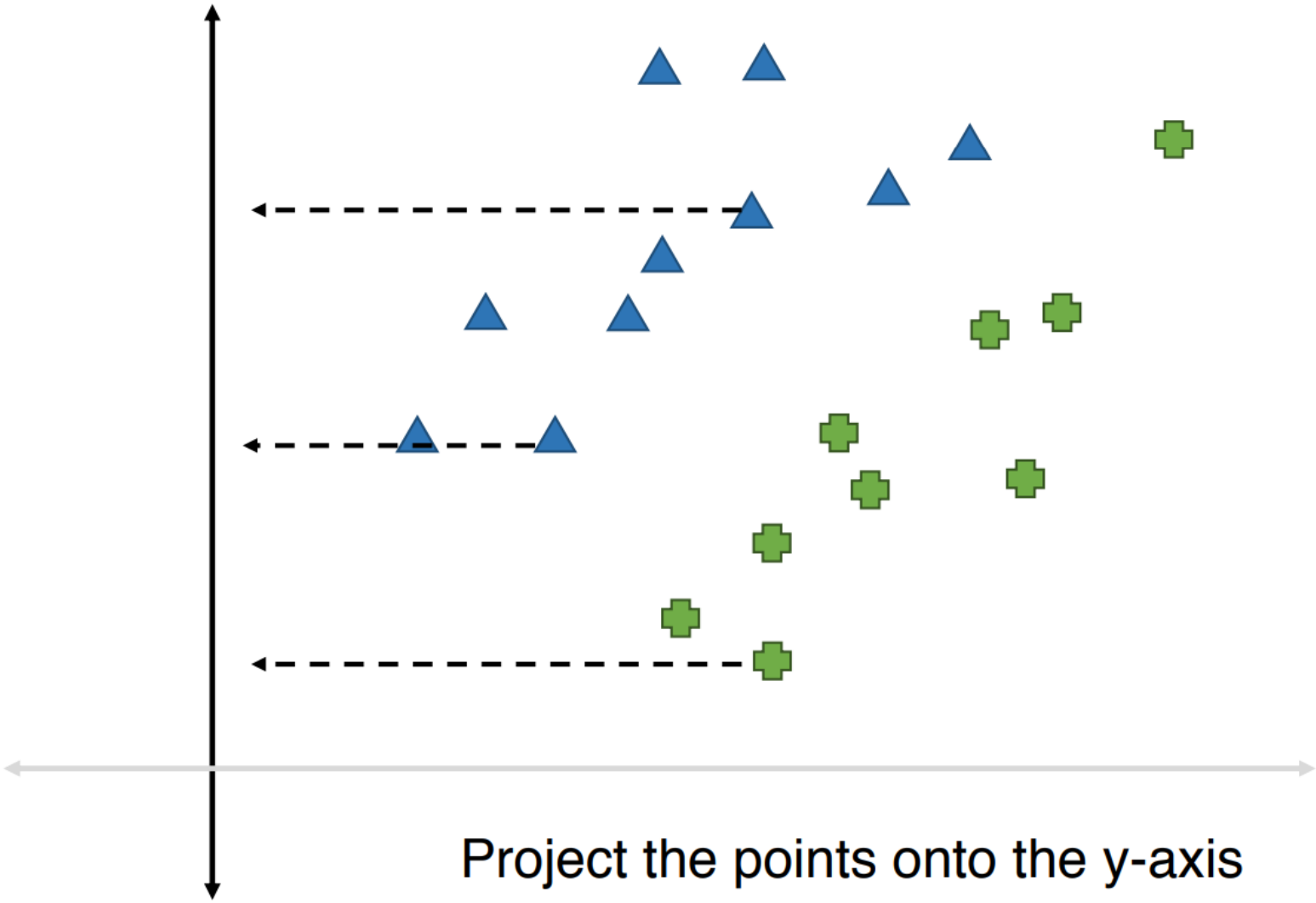
Geometric Intuition

Suppose we chose the feature represented by the x-axis



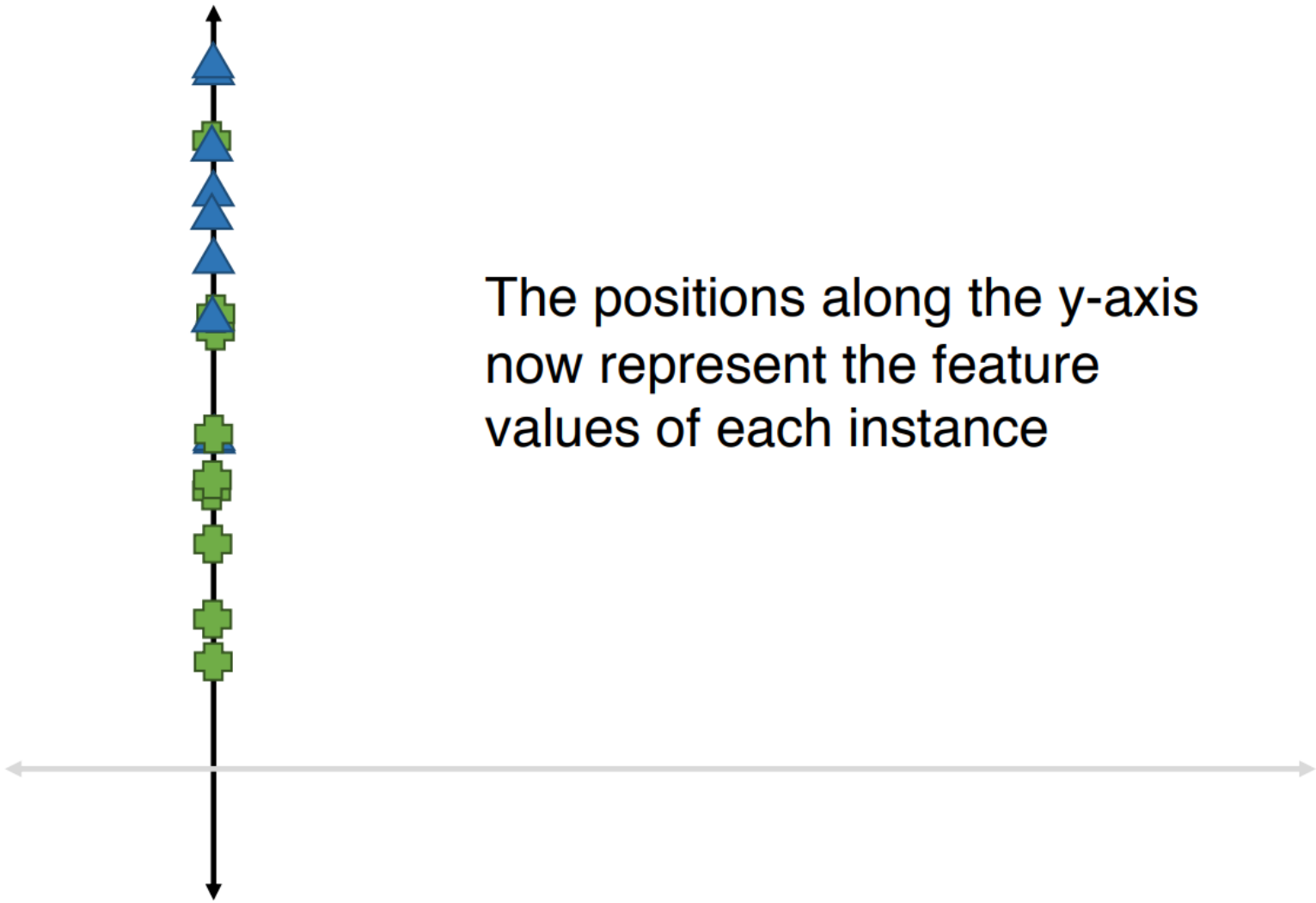
Geometric Intuition

Suppose we choose the feature represented by the y-axis



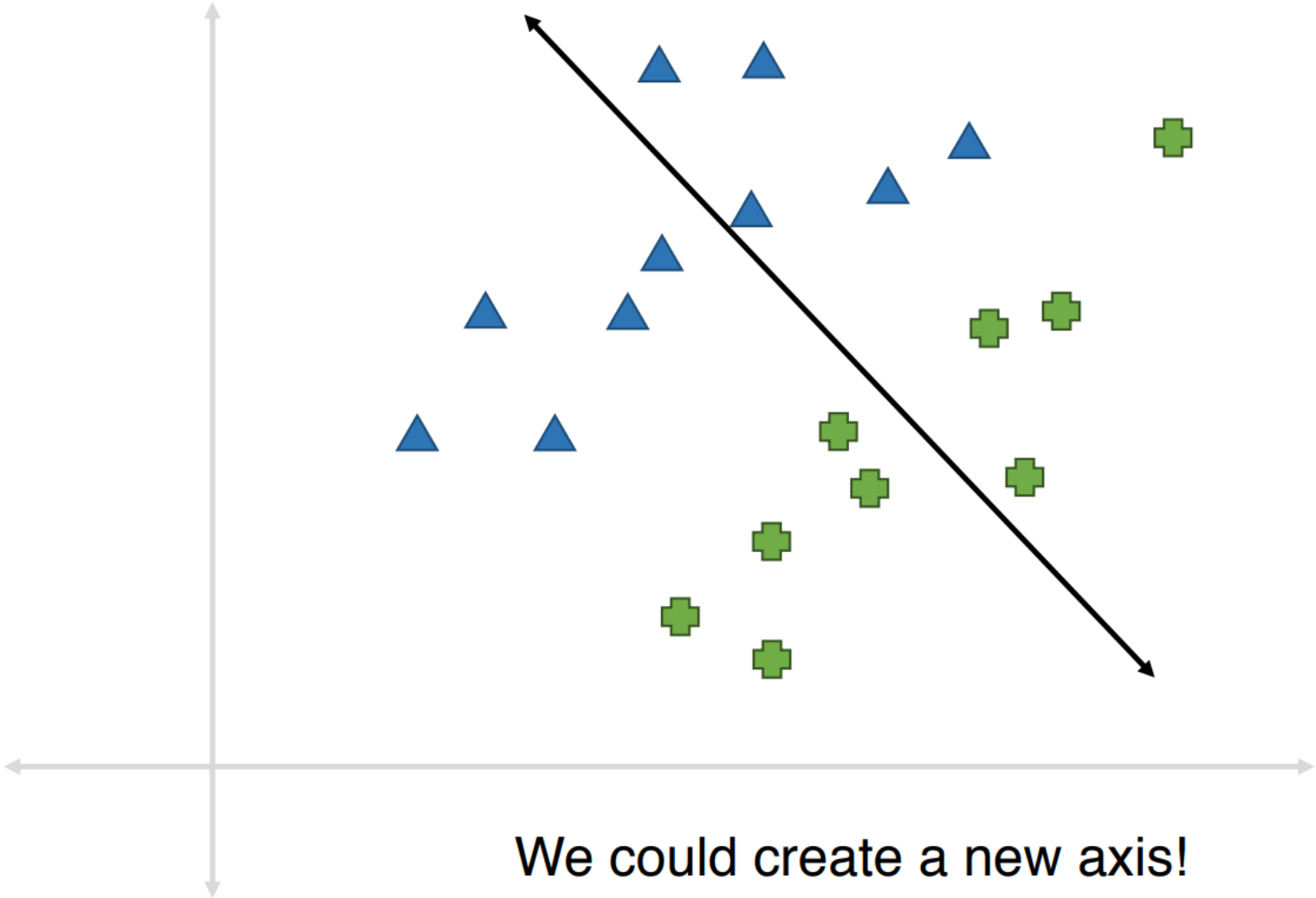
Geometric Intuition

Suppose we choose the feature represented by the y-axis



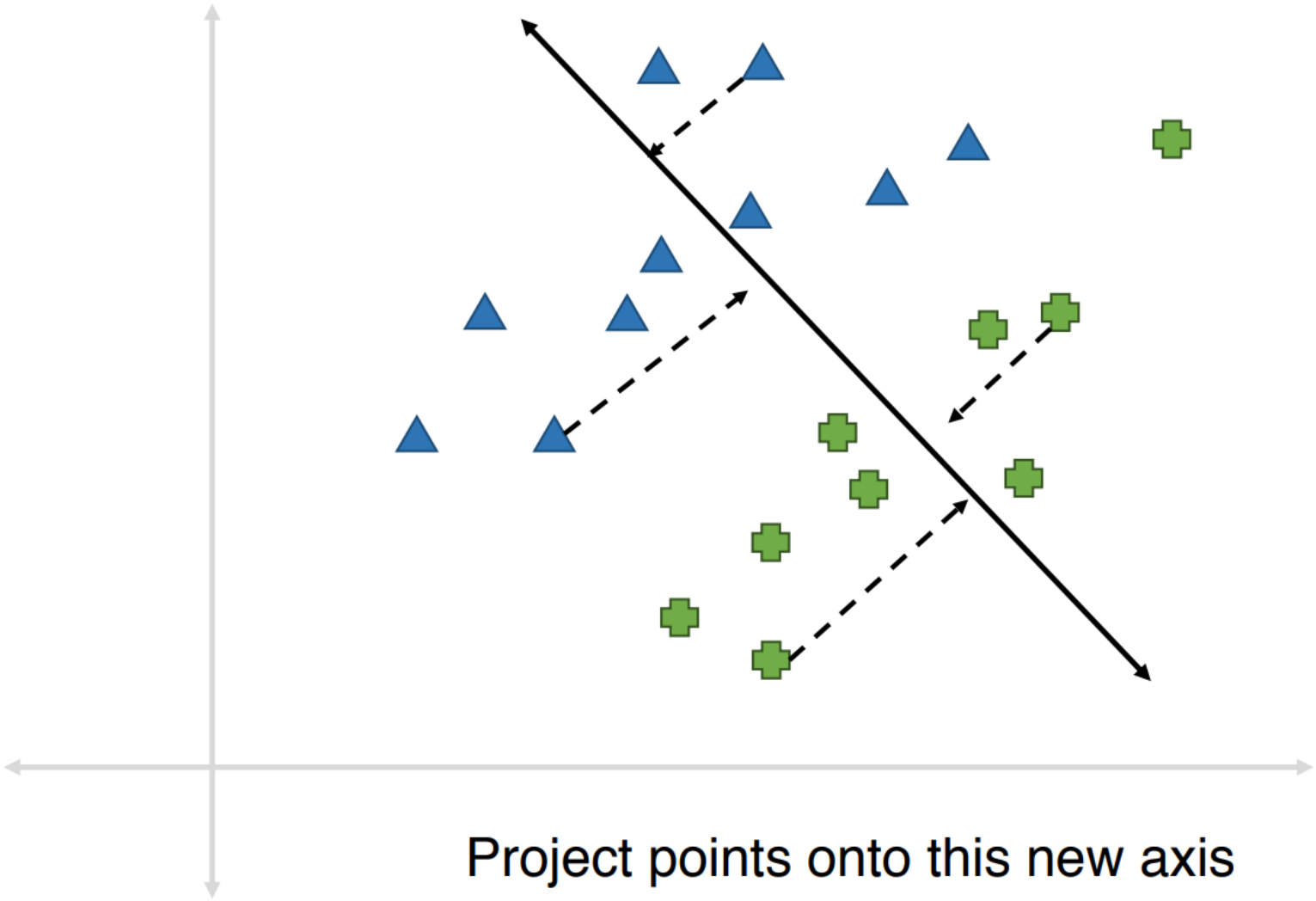
Geometric Intuition

We don't have to restrict ourselves to picking either the x-axis or the y-axis



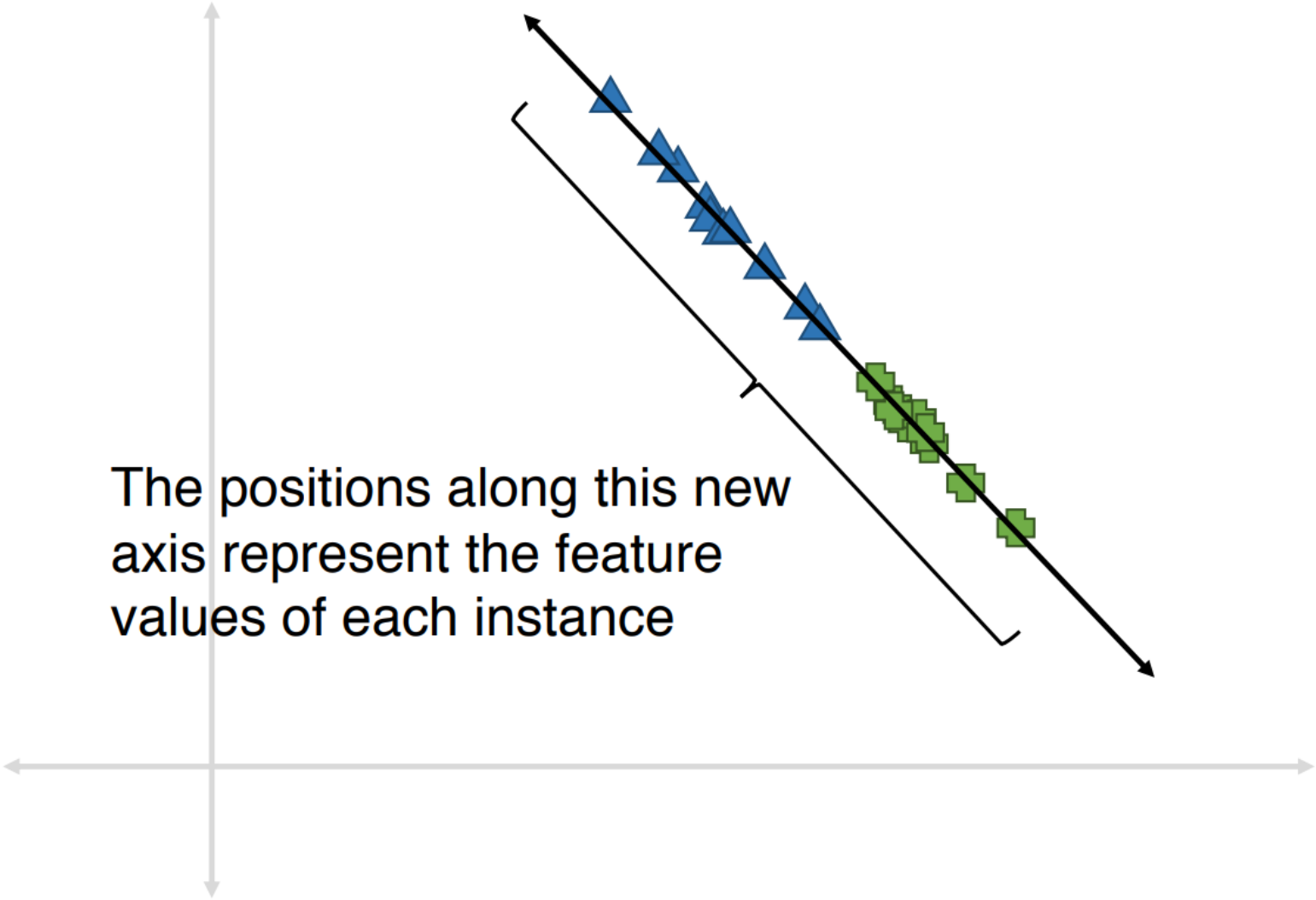
Geometric Intuition

We don't have to restrict ourselves to picking either the x-axis or the y-axis



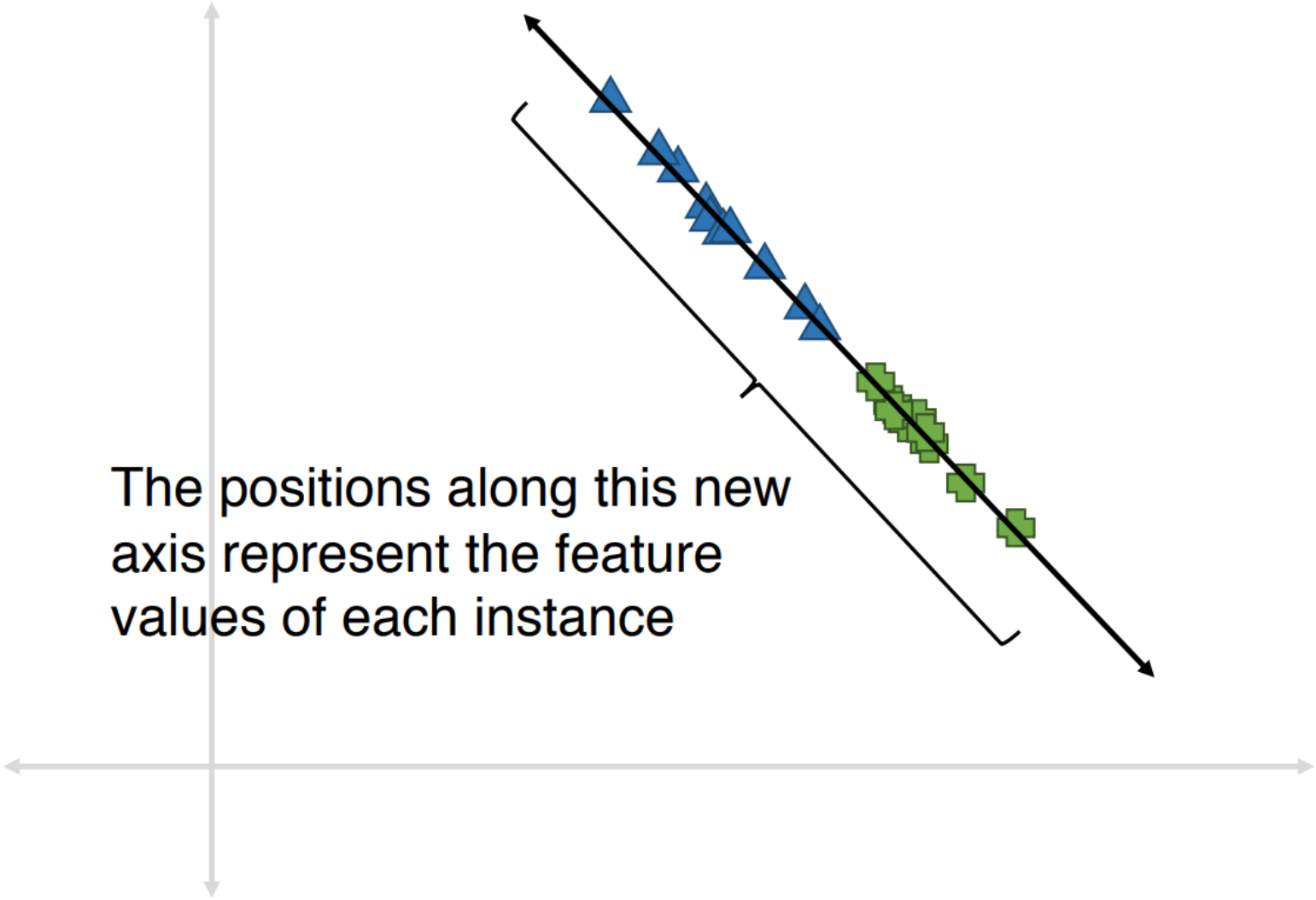
Geometric Intuition

We don't have to restrict ourselves to picking either the x-axis or the y-axis



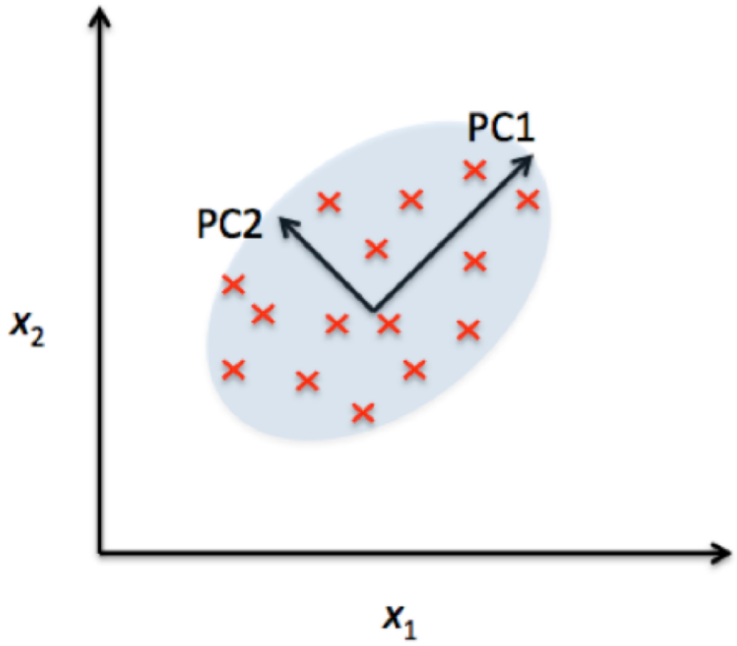
Geometric Intuition

This is an example of **transforming** the feature space (as obbosed to **selecting** a subset of features)



Principle Component Analysis

Principle component analysis (**PCA**) is a widely used technique that chooses new axes to project the data onto



The new axes are called the **principle components**

Principle Component Analysis

PCA does not use any information about the class labels

- **Unsupervised** dimensionality reduction
- This has advantages and disadvantages

So, how does PCA decide how to choose axes?

- Basic idea: pick an axis so that the values will have **high variance** once projected onto it

Principle Component Analysis

Overview of PCA algorithm:

- Start by identifying the principle component (axis) with the highest variance
- Pick another principle component that is orthogonal (forms a right angle with) the previous component(s) with the next-highest variance.
  - Why orthogonal? Don't want to pick another nearly identical component, or it won't give you much information beyond what the other component is already providing
  - Point is to reduce redundancy
- Keep repeating until K principle components have been chosen.

Principle Component Analysis

Overview of PCA algorithm:

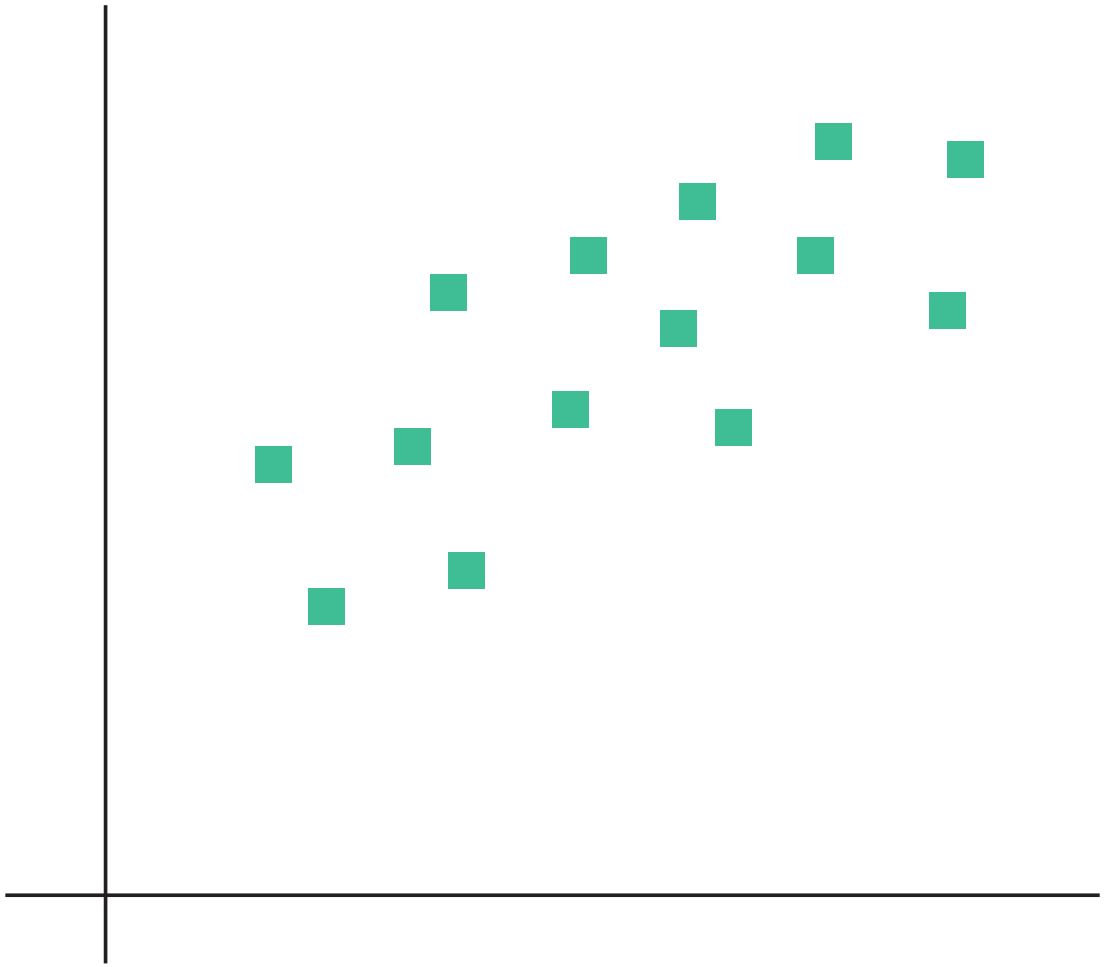
- The algorithm will also learn a function that transforms an instance **x** into the projected space.
- Then you can use the transformed instances in your prediction algorithm

Note: variance depends on the scale of the values

- For PCA to work, important that all features are on the same scale
- Perform **normalization/standardization** first

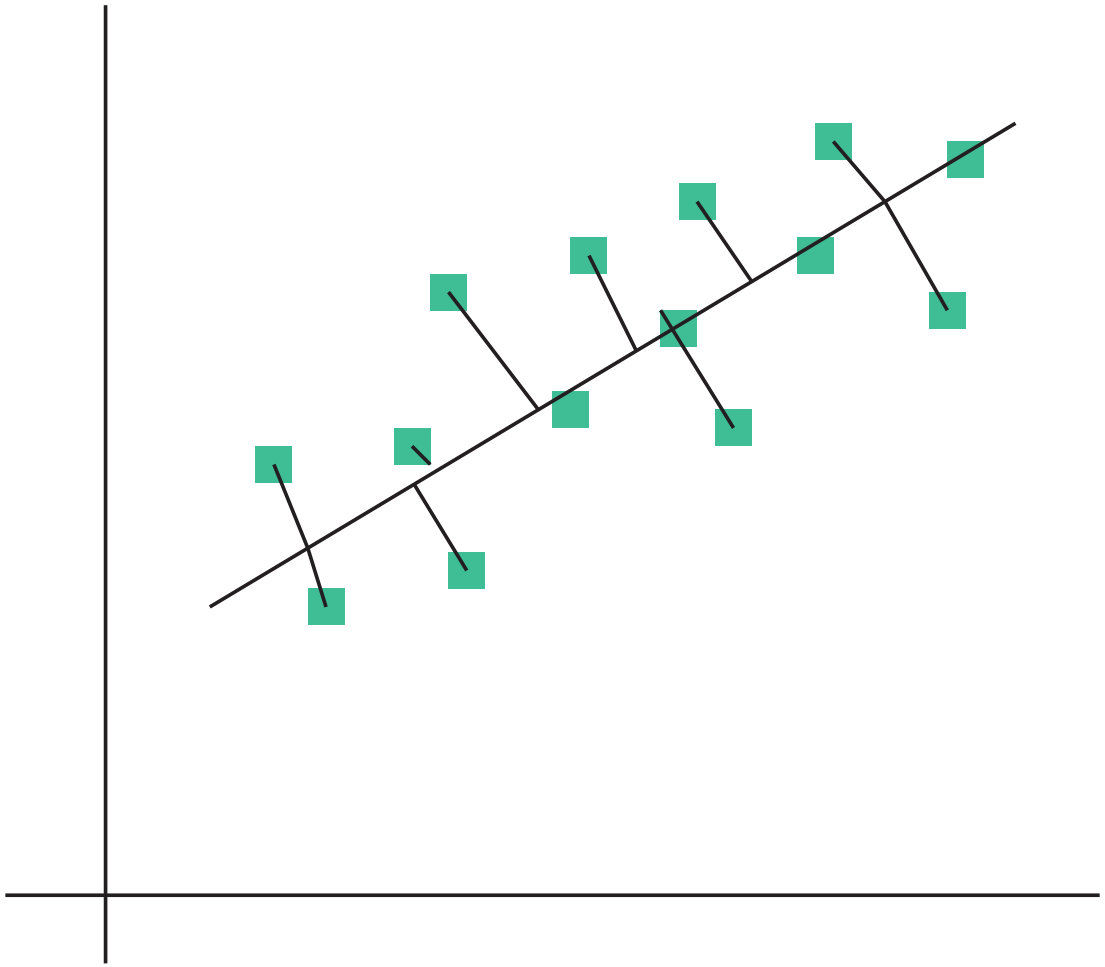
Geometric Intuition

Suppose we have two dimensions (two features)



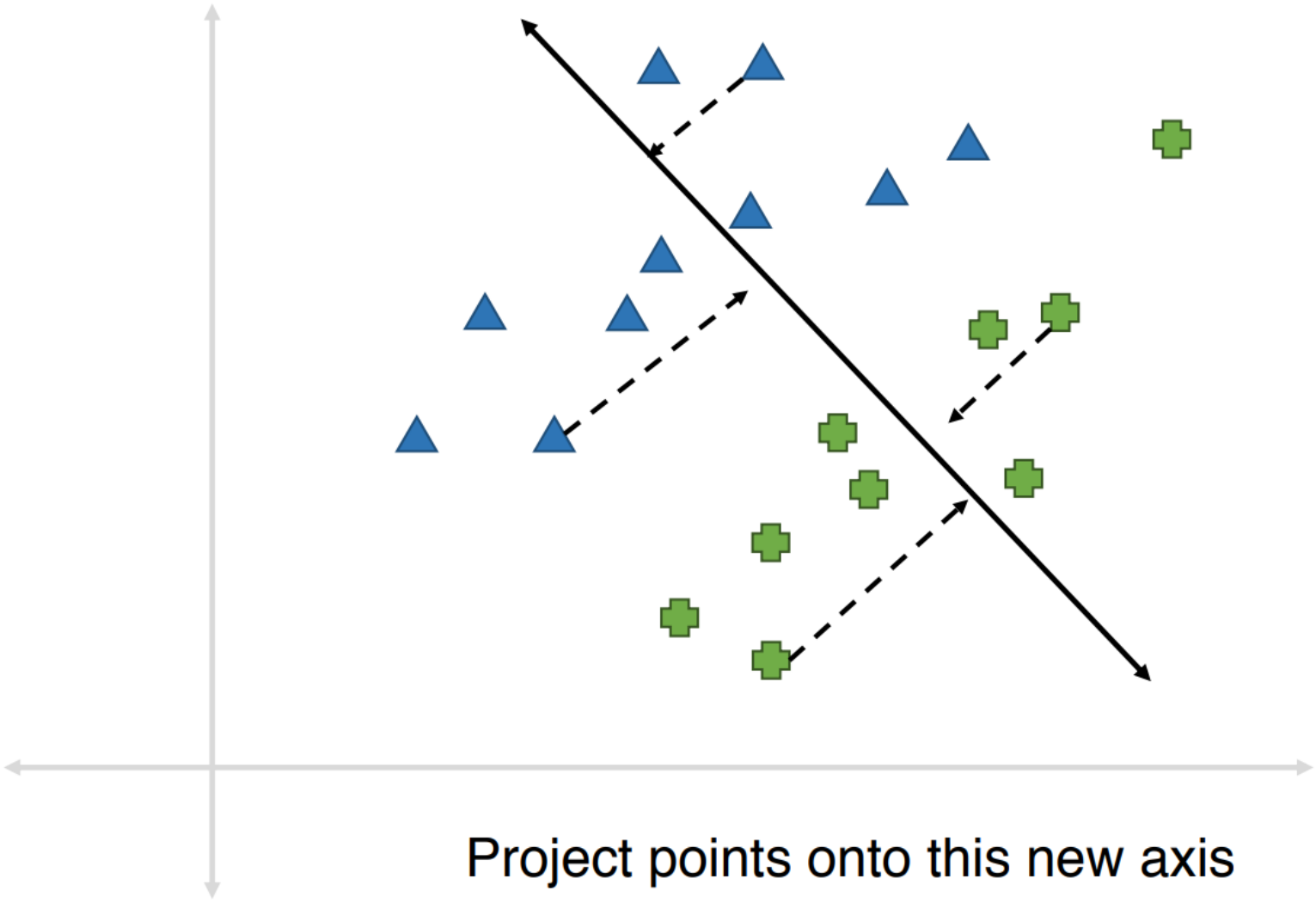
Geometric Intuition

Suppose we have two dimensions (two features)



Geometric Intuition

Depending on the datapoints are distributed, this could now work well



PCA with Scikit-learn

```
>>> X, y = # Assume we have already loaded data
>>> from sklearn.decomposition import PCA
>>> from sklearn.preprocessing import StandardScaler
>>> ss = StandardScaler()
>>> X_scale = ss.fit_transform(X)
>>> pca = PCA(n_components=2)
>>> X_scale.shape
(50, 200)
>>> new_X = pca.fit_transform(X_scale)
>>> new_X.shape
(50, 2)
```

Exercise 3

For this exercise use the iris dataset, then apply PCA with 2 dimensions and plot the principal components (plotting code is provided in the notebook)

Missing Data

Missing data is information that we want to know, but don't!

It can com in many forms:

- People not answering questions on surveys
- Inaccurate recordings of the height of plants that need to be discarded
- Canceled runs in a driving experiment due to rain

Types of Missing Data

- Missing Completely at Random (MCAR)
  - P(missing) is unrelated to the process under study
- Missing at Random (MAR)
  - P(missing) depends only on **observed data**
- Missing Not at Random (MNAR)
  - P(missing) depends on both observed and **unobserved data**.

Type type of missing data drastically affects what we can ultimately do to compensate for missing-ness

Complete Case Analysis

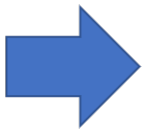
Delete all rows with **any missing values** at all, so you are left only with observations where all variables are observed.

This is the **easiest way to handle missing data**. In some cases it will work fine; **in others, ????**

- Loss of sample will lead to variance larger than reflected by the size of your data
- May bias your sample

Complete Case (row) Analysis

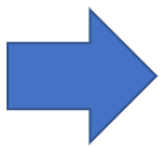
Credit	Term	Income	y
Excellent	3 yrs	High	safe
Fair	?	Low	Risky
Fair	3 yrs	High	Safe
Poor	5 yrs	High	Risky
Excellent	3 yrs	Low	Risky
Fair	5 yrs	High	Safe
Poor	3 yrs	Low	Risky
Poor	?	Low	Safe
Fair	?	High	Safe



Credit	Term	Income	y
Excellent	3 yrs	High	safe
Fair	3 yrs	High	Safe
Poor	5 yrs	High	Risky
Excellent	3 yrs	Low	Risky
Fair	5 yrs	High	Safe
Poor	3 yrs	Low	Risky

Complete Case (row) Analysis

Credit	Term	Income	y
Excellent	3 yrs	High	safe
Fair	?	Low	Risky
Fair	3 yrs	High	Safe
Poor	5 yrs	High	Risky
Excellent	3 yrs	Low	Risky
Fair	5 yrs	High	Safe
Poor	3 yrs	Low	Risky
Poor	?	Low	Safe
Fair	?	High	Safe



Credit	Income	y
Excellent	High	safe
Fair	Low	Risky
Fair	High	Safe
Poor	High	Risky
Excellent	Low	Risky
Fair	High	Safe
Poor	Low	Risky
Poor	Low	Safe
Fair	High	Safe

Missing Value Skipping: Pros and Cons

Pros

- Easy to understand and implement
- Can be applied to any model (decision trees, logistic regression, linear regression, ...)

Cons

- Removing data points and features may remove important information from the data
- Unclear when it's better to remove data points versus features
- Doesn't help if data is missing at test time

Removing Missing Rows - Numpy

example.py

```
import numpy as np
a = np.array([
        [1, 0, 0],
        [0, np.nan, 0],
        [0, 0, 0],
        [np.nan, np.nan, np.nan],
        [2, 3, 4]
])
print(a.shape)
mask = np.isnan(a).any(axis=1)
print(mask)
print(a[ mask].shape)
```

anthony@MacBook:~$ python example.py

```
(5, 3)
[False True False True False]
(3, 3)
```

Removing Missing Columns - Numpy

example.py

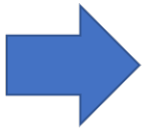
```
import numpy as np
a = np.array([
        [1, 0, 0],
        [0, np.nan, 0],
        [0, 0, 0],
        [np.nan, np.nan, 1],
        [2, 3, 4]
])
print(a.shape)
mask = np.isnan(a).any(axis=0)
print(mask)
print(a[:, mask].shape)
```

anthony@MacBook:~$ python example.py

```
(5, 3)
[True True False]
(5, 1)
```

Mode

Credit	Term	Income	y
Excellent	3 yrs	High	safe
Fair	?	Low	Risky
Fair	3 yrs	High	Safe
Poor	5 yrs	High	Risky
Excellent	3 yrs	Low	Risky
Fair	5 yrs	High	Safe
Poor	3 yrs	Low	Risky
Poor	?	Low	Safe
Fair	?	High	Safe



Credit	Term	Income	y
Excellent	3 yrs	High	safe
Fair	**3 yrs**	Low	Risky
Fair	3 yrs	High	Safe
Poor	5 yrs	High	Risky
Excellent	3 yrs	Low	Risky
Fair	5 yrs	High	Safe
Poor	3 yrs	Low	Risky
Poor	**3 yrs**	Low	Safe
Fair	**3 yrs**	High	Safe

Impute each feature with missing values:

- **Categorical features use mode**: Most popular value (mode) of non-missing x_i
- **Numerical features use average or median**: Average or median value of non-missing x_i

There are other methods, e.g., expectation-maximization algorithm, regressing/classifying on missing columns. See:

`https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html`

Missing Value Imputation: Pros and Cons

Pros

- Easy to understand and implement
- Can be applied to any model (decision trees, logistic regression, linear regression, ...)
- Can be used at prediction time using the same imputation rules

Cons

- May result in systematic errors
  - Example: Feature "age" missing in all banks in Washington by state law

Imputation with Scikit-learn

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
```

If you have an old version of sklearn installed use the following instead

```
>>> from sklearn.preprocessing import Imputer as SimpleImputer
>>> imp_mean = SimpleImputer(missing_values=np.nan,
strategy='mean')
>>> imp_mean.fit([[7, 2, 3], [4, np.nan, 6], [10, 5, 9]])
>>> X = [[np.nan, 2, 3], [4, np.nan, 6], [10, np.nan, 9]]
>>> print(imp_mean.transform(X))
[ 7.   2.   3. ]
[ 4.   3.5  6. ]
[10.   3.5  9. ]
```

Exercise 4

Using the "prima-indians-diabetes.csv" (the last column is the class, 1 or 0) dataset compare the F1 score of the following missing data strategies:

- Remove all columns with missing values
- Impute using the mean value.
- Impute using the median

What is the best missing data strategy for this dataset?

Note: Missing values are set to 0 in this dataset, you will need to convert them to np.nan, and only the following columns can have missing values: 2, 3, 4, 5, 6.