Lab 4

K-NN

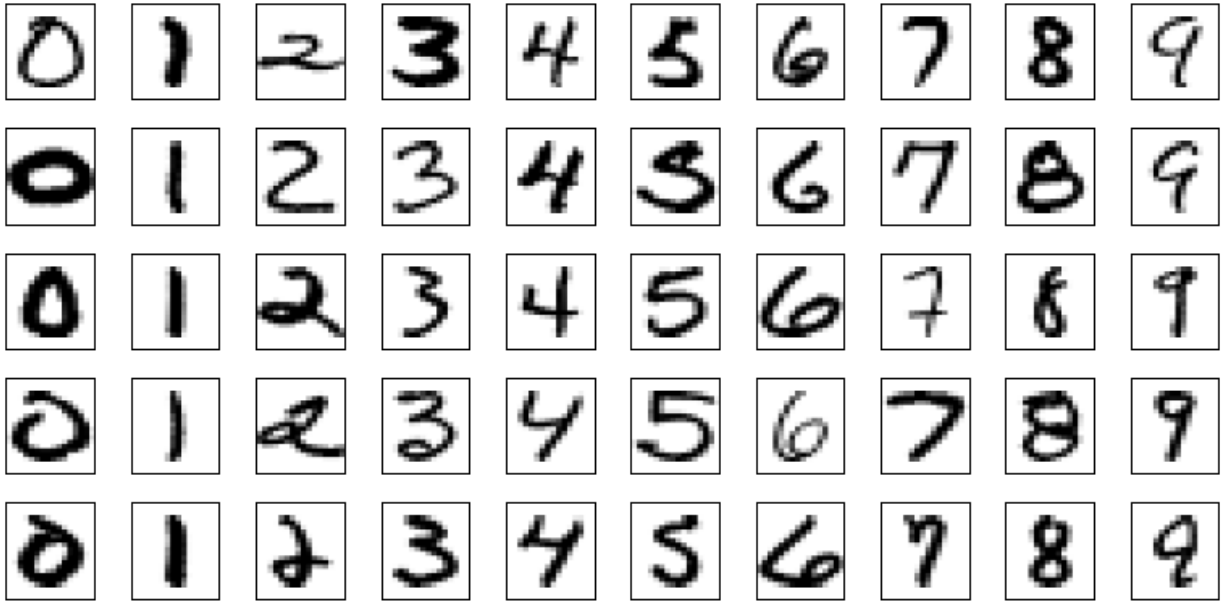

UPC
telecom BCN
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Introduction

- Objectives
  - Use a non-parametric classifier, k-nearest neighbors.
  - Test this method on an image dataset of handwritten digits
  - Use a simple grid search cross-validation for hyperparameter selection
  - Learn how to use scikit-learn pipelines

ZIP dataset

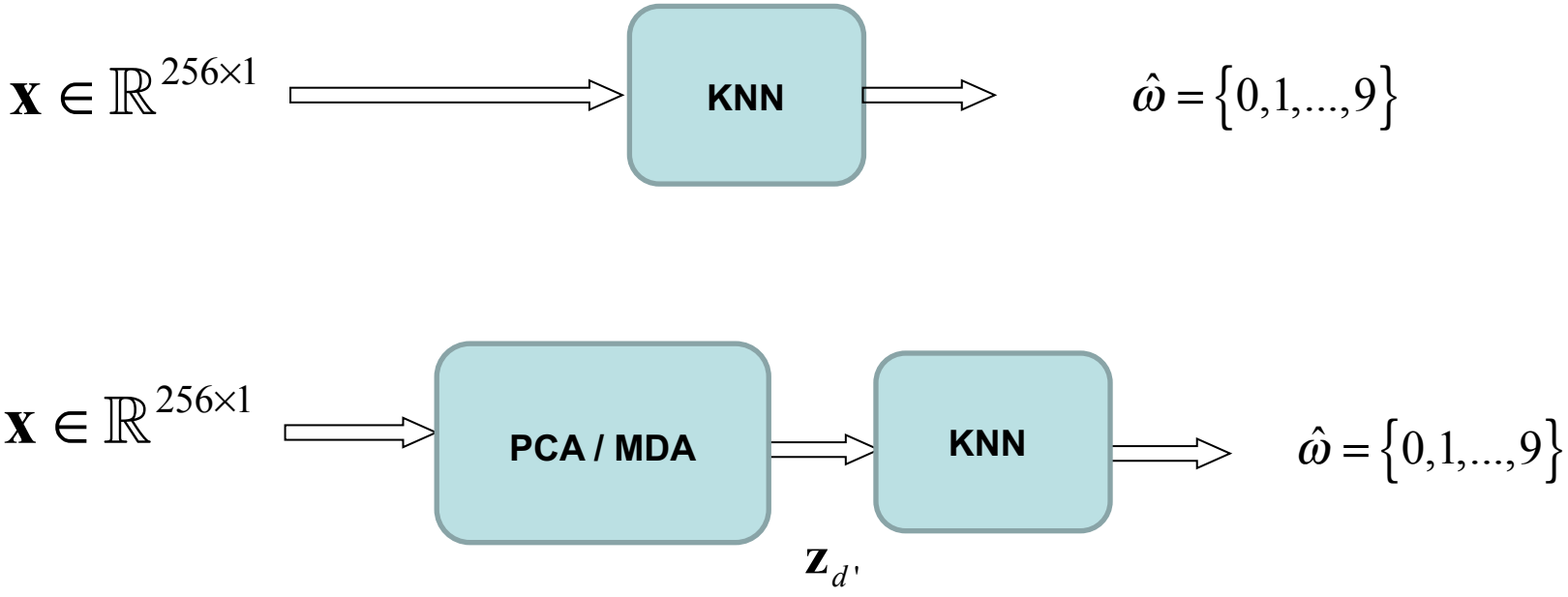
- Each vector of d=256 features represents an image of size 16x16 (read by raws) from one of ten classes corresponding to handwritten digits 0 to 9.
- **Parameters**
  - total number of vectors in the Training dataset: 7291
  - total number of vectors in the Test dataset: 2007
  - initial dimension of the vectors d=256
  - number of classes c=10



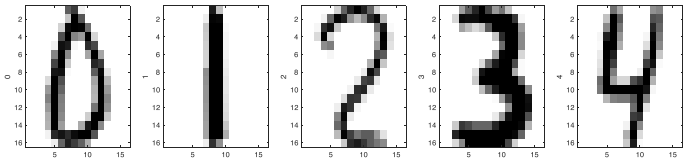
k-nearest neighbors

For each test vector find the ”k” closest training vectors and predict the most popular class among the K training vectors (majority voting)

Example: ZIP dataset. There are 10 classes (digits 0 to 9). Vectors to classify are images of handwritten digits

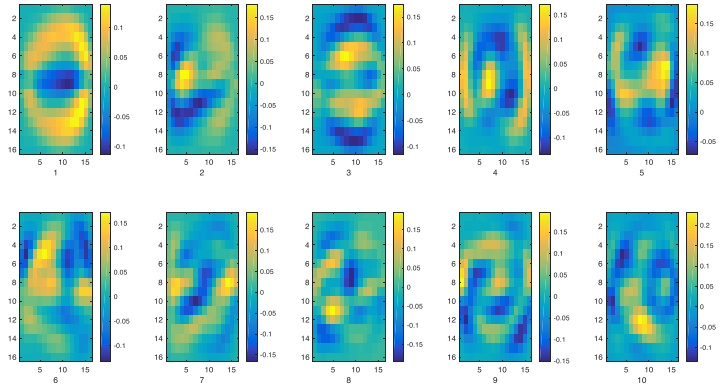


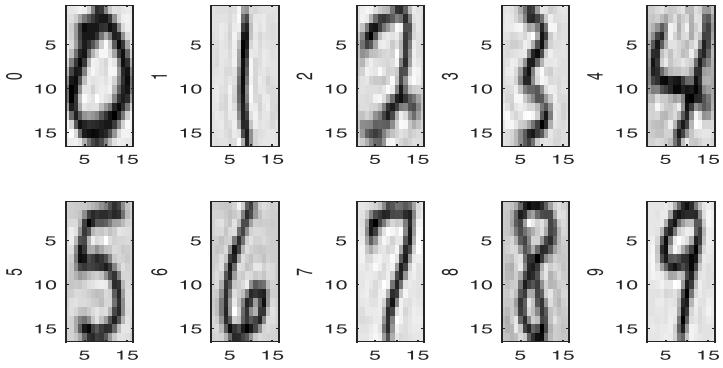
Example: dimensionality reduction from 256 to 64 features using PCA



Sample original images

Eigenimages

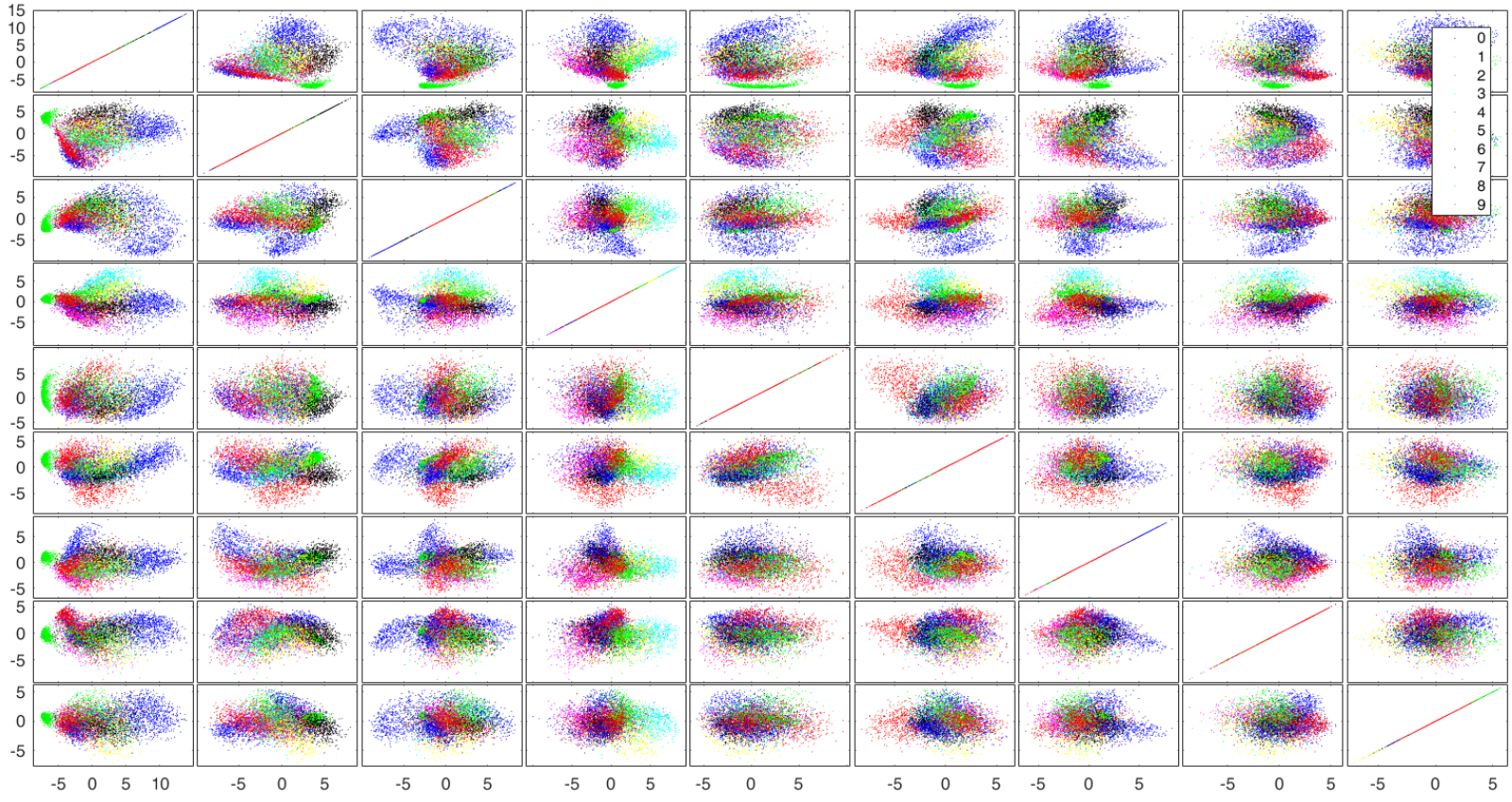




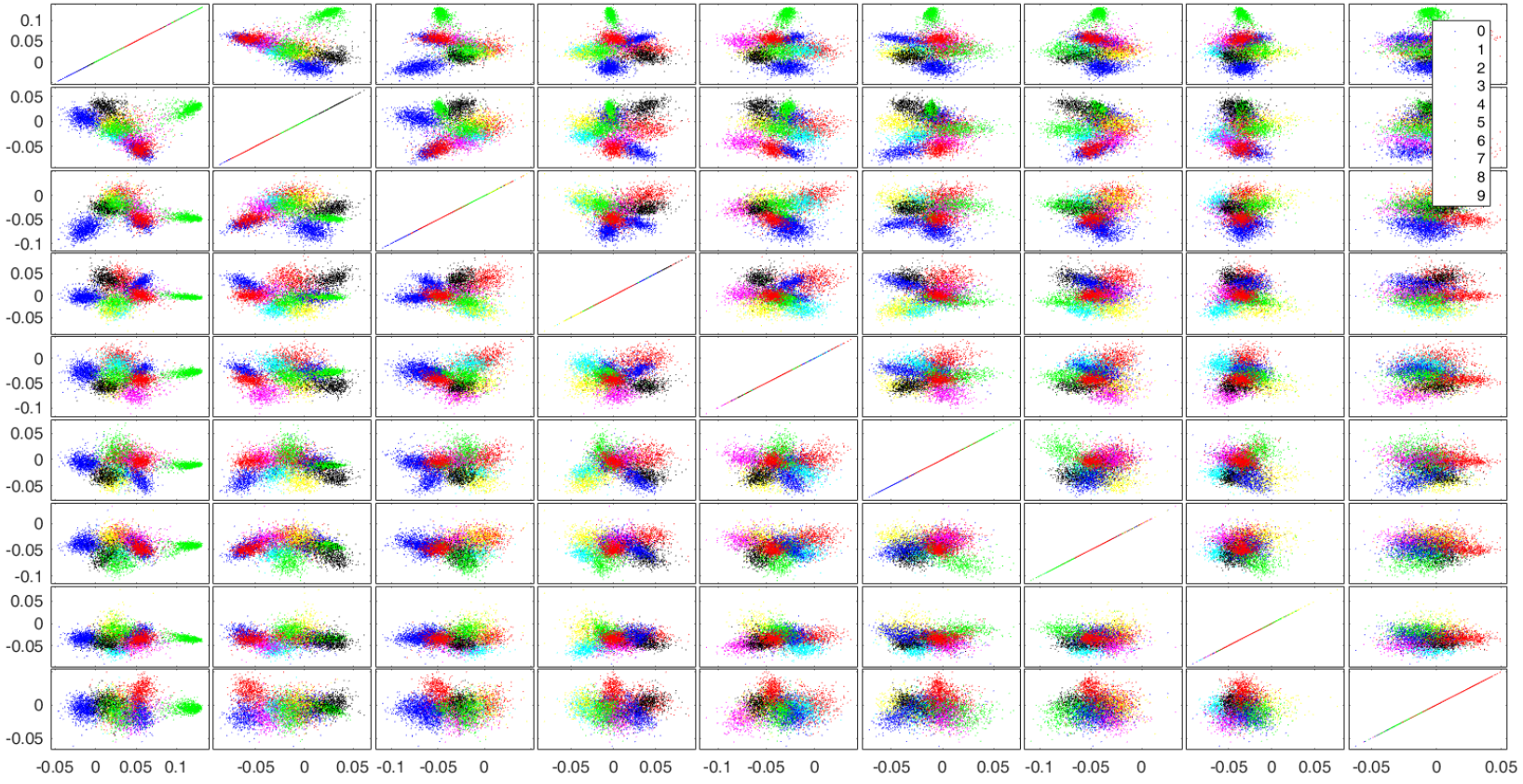
Sample reconstructed images

Can we still recognize digits using the reconstructed images?

Example: scatter plot using PCA (9 features)



Example: scatter plot using MDA (9 features)



kNN in scikit-learn

- *class* `sklearn.neighbors.`**`KNeighborsClassifier`**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None*)
- **n_neighbors** *int, default=5* Number of neighbors to use
- **Weights***:* Weight function used in prediction. Possible values:

  'uniform': uniform weights. All points in each neighborhood are weighted equally.

  'distance' : weight points by the inverse of their distance.

  [callable] : a user-defined function
- **Algorithm***:* Algorithm used to compute the nearest neighbors (N samples, D features):

  'brute' will use a brute-force search (for small N)

  'kd_tree' will use KDTree (for large N and small D, retrieval in O(log(N))

  'ball_tree' will use BallTree (for large N and large D)

  'auto' decides the most appropriate algorithm based on the values passed to fit
- **Metric:** Metric to use for distance computation. Default is "minkowski"

Pipelines in scikit-learn

- Scikit-learn provides a **Pipeline utility** to help automate machine learning workflows.

- A Pipeline can be used to chain multiple estimators into one. This is useful as there is often a fixed sequence of steps in processing the data, for example feature selection, normalization and classification.

- Purposes
  - **Convenience and encapsulation** You only have to call fit and predict once on your data to fit a whole sequence of estimators.
  - **Joint parameter selection** You can grid search over parameters of all estimators in the pipeline at once.
  - **Safety** Pipelines help avoid leaking statistics from your test data into the trained model in cross-validation, by ensuring that the same samples are used to train the transformers and predictors.

- All estimators in a pipeline, except the last one, must be transformers (i.e. must have a transform method). The last estimator may be any type (transformer, classifier, etc.).

Pipelines in scikit-learn

We wil define a pipeline that

1. Standardizes features by removing the mean
2. Reduces dimensionality and
3. Applies a kNN classifier

```
model = Pipeline([
('center', StandardScaler(with_mean=True, with_std=False)),
('reduce_dim', PCA(n_components=10)),
('clf', KNeighborsClassifier())
])
```

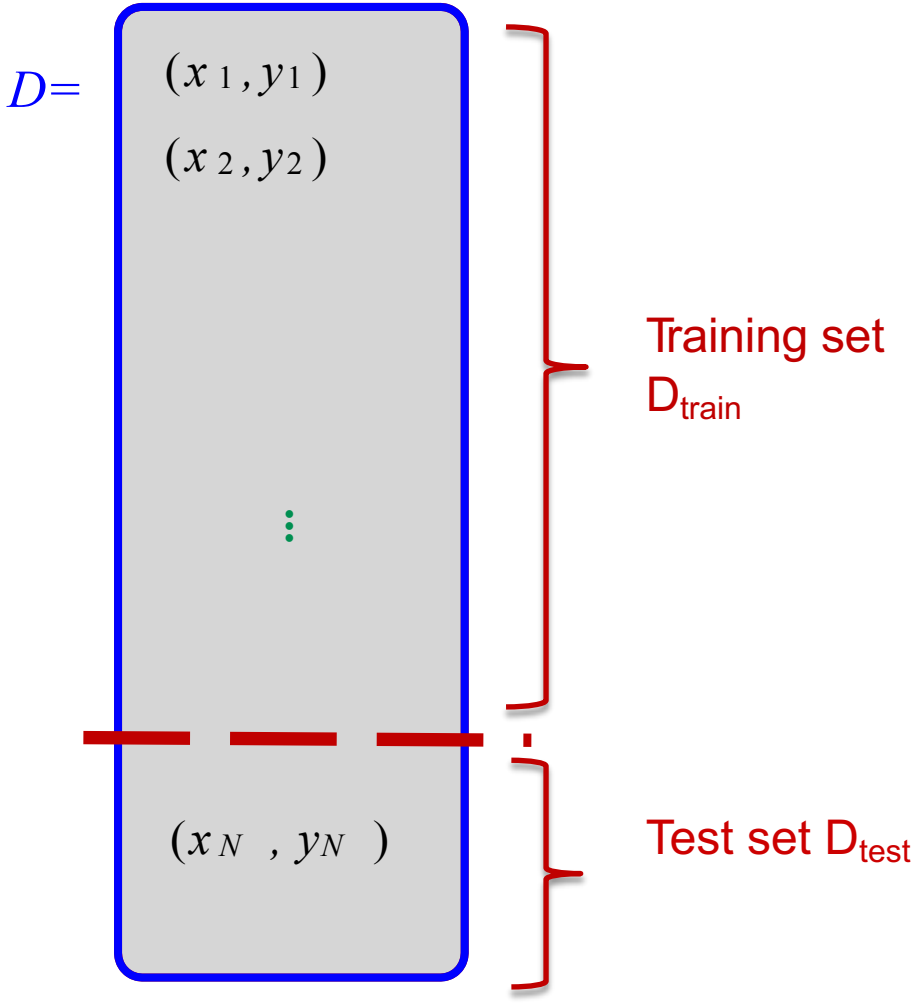
Fit all the transformers one after the other, and fit the classifier:

```
model.fit(X_train, y_train)
```

Apply all the transforms and finaly calls the predict of the last estimator:

```
pred_train = model.predict(X_train)
pred_test = model.predict(X_test)
```

Simple train-test procedure



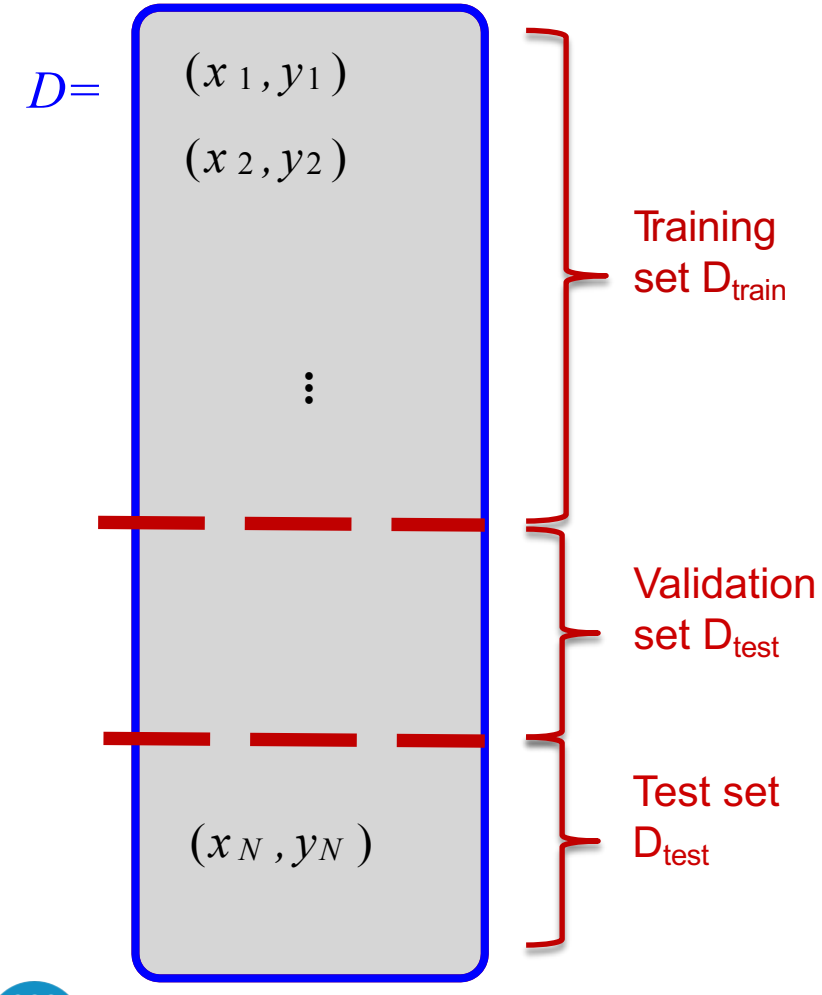
If there are no hyperparameters!!

- Provided large enough dataset D drown from p_{data}
- Arrange samples in random order
- Split dataset in two: D_{train} and D_{test}
- Use D_{train} to find the best predictor f
- Use D_{test} to evaluate the generalization performance of f

Hyperparameter tuning: train-val-test set

Make sure examples are in random order

Split data D in 3: D_{train} D_{valid} D_{test}



- Data is split into 3 subsets
  - Training and Validation set used only to find the right predictor (**optimize hyperparameters**)
    - Repeat training on D_{train} and evaluation on D_{valid} **for each value of hyperparameter**
    - Select the best performing value of hyperp.
    - Retrain the model using training + validation data using the best hyperparameter
  - A Test set used to report the performance of the algorithm
- The sets must be disjoint

Parameter search: GridSearchCV

- **GridSearchCV exhaustively considers all parameter combinations**
- The GridSearchCV instance implements the usual estimator API: when "fitting" it on a dataset, all the possible combinations of parameter values are evaluated and the best combination is retained. A GridSeachCV object internally iterates over a parameter grid and computes cross-validated scores for each hyper-parameter set.
- **Main parameters:** estimator (KNN), param_grid (n_neighbors and possible values), cross validation strategy (here just one split)
- **Score function:** by default, it uses the score function of the estimator: accuracy_score for classification

```
model2 = Pipeline([
('center', StandardScaler(with_mean=True, with_std=False)),
('reduce_dim', dim_reducer),
('clf', KNeighborsClassifier())])

Ks = [1, 2, 3, 4, 5, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15]

grid_search_cv = GridSearchCV(estimator = model2,
param_grid={"clf__n_neighbors": Ks},
cv=StratifiedShuffleSplit(n_splits=1, train_size=0.75,random_state=1),
return_train_score=True)
grid_search_cv.fit(X_train, y_train)
pred_train = grid_search_cv.predict(X_train)
pred_test = grid_search_cv.predict(X_test)
```

Feature scaling

- Feature scaling is a method used to standardize the range of independent variables or features of data.

- Why it is important?
  - If range of inputs varies, in some algorithms, object functions will not work properly.
  - Gradient descent converges much faster with feature scaling done. Gradient descent is a common optimization algorithm used in logistic regression, SVMs, neural networks etc.
  - Algorithms that involve distance calculation like KNN, Clustering are also affected by the magnitude of the feature.

Note: Tree-based algorithms are almost the only algorithms that are not affected by the magnitude of the input

Feature scaling

How to perform feature scaling? sklearn.preprocessing package

	Method	Definition	Pros	Cons
StandardScaler	Normalization - Standardization (Z-score scaling)	removes the mean and scales the data to unit variance. z = (X - X.mean) / std	feature is rescaled to have a standard normal distribution that centered around 0 with SD of 1	compress the observations in the narrow range if the variable is skewed or has outliers, thus impair the predictive power.
MinMaxScaler MaxAbsScaler	Min-Max scaling	transforms features by scaling each feature to a given range. Default to [0,1]. X_scaled = (X - X.min / (X.max - X.min)	/	compress the observations in the narrow range if the variable is skewed or has outliers, thus impair the predictive power.
RobustScaler	Robust scaling	removes the median and scales the data according to the quantile range (defaults to IQR) X_scaled = (X - X.median) / IQR	better at preserving the spread of the variable after transformation for skewed variables	/