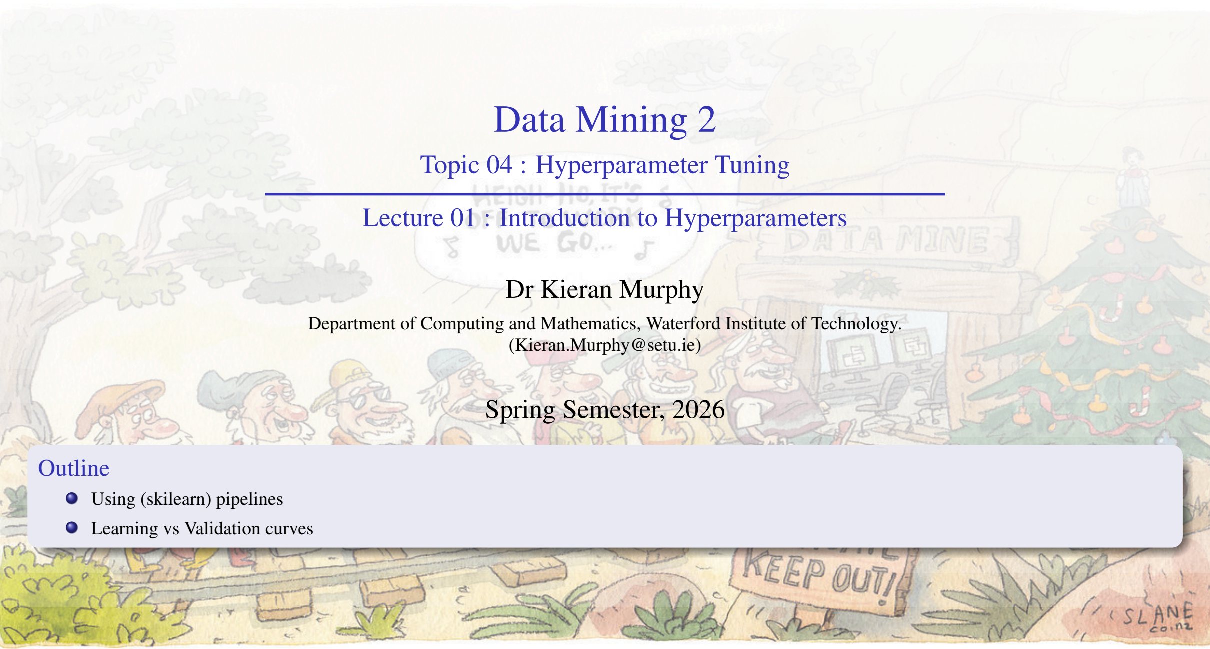


Data Mining 2

Topic 04 : Hyperparameter Tuning

Lecture 01 : Introduction to Hyperparameters

Dr Kieran Murphy

Department of Computing and Mathematics, Waterford Institute of Technology.
(Kieran.Murphy@setu.ie)

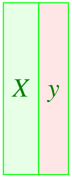
Spring Semester, 2026

Outline

- Using (skilearn) pipelines
- Learning vs Validation curves

Outline

The Problem

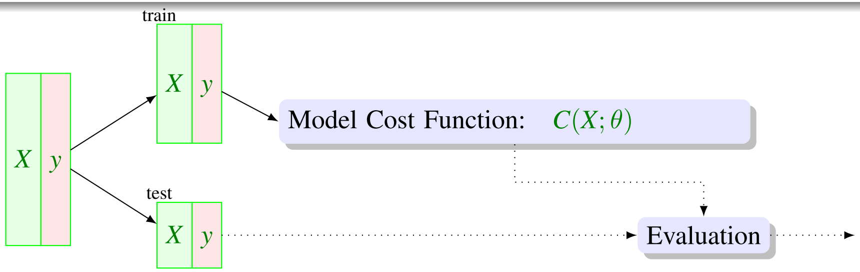


Model Cost Function: $C(X;\theta)$

- To date our model training has focused on optimising the model parameters, but a model can have additional parameters (called hyperparameters) whose values can have significant impact on the performance of the model.
- Usually cannot estimate the hyperparameters as part of the learning step because:
  - Hyperparameters play a quantifiable different role — think, degree of polynomial vs polynomial coefficients.
  - If not continuous — think L_1 vs L_2 option — then can't be used in gradient optimisation methods.
  - Greatly increase the complexity of the learning process.

$\Rightarrow$ Need a separate step to determine optimal values for the hyperparameters.

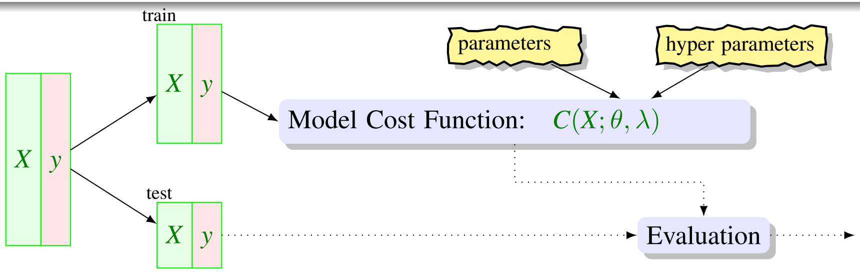
The Problem



- To date our model training has focused on optimising the model parameters, but a model can have additional parameters (called hyperparameters) whose values can have significant impact on the performance of the model.
- Usually cannot estimate the hyperparameters as part of the learning step because:
  - Hyperparameters play a quantifiable different role — think, degree of polynomial vs polynomial coefficients.
  - If not continuous — think L_1 vs L_2 option — then can't be used in gradient optimisation methods.
  - Greatly increase the complexity of the learning process.

$\Rightarrow$ Need a separate step to determine optimal values for the hyperparameters.

The Problem



- To date our model training has focused on optimising the model parameters, but a model can have additional parameters (called hyperparameters) whose values can have significant impact on the performance of the model.
- Usually cannot estimate the hyperparameters as part of the learning step because:
  - Hyperparameters play a quantifiable different role — think, degree of polynomial vs polynomial coefficients.
  - If not continuous — think L_1 vs L_2 option — then can't be used in gradient optimisation methods.
  - Greatly increase the complexity of the learning process.

$\Rightarrow$ Need a separate step to determine optimal values for the hyperparameters.

Parameters vs Hyperparameters

Parameters

- Weights/coefficients/numbers learnt during the training process.
- Automatically estimated.
- Examples:
  - coefficients in a linear / logistic regression.
  - support vectors in a support vector machine.
  - weights in an artificial neural network.

Hyperparameters

- ‘Knobs’/‘dials’/’switches’ used to control the training process.
- Manually specified/set.
- Often used in processes to help estimate model parameters.
- Often set using heuristics.
- Examples:
  - Model selection
  - Feature selection
  - Size/depth of neural networks

Advances in machine learning techniques result in hyperparameters becoming parameters as algorithms improve and computational power increases.

Example — Ridge/Lasso Regression

Both regression methods introduce a new hyperparameter[*], λ, that controls the importance of a penalty term:

Ridge Cost function

$$C(X;\theta,\lambda) = \left\|(\theta_0 + \theta_1\vec{x}_1 + \cdots + \theta_n\vec{x}_n) - \vec{y}\right\|_2^2 + \lambda\left\|\theta\right\|_2^2$$

- Performs L_2 regularisation, i.e., adds penalty equivalent to square of the magnitude of coefficients.
- Larger values of λ helps overfitting and effects of multi-collinearity in X.

Lasso Cost function

$$C(X;\theta,\lambda) = \left\|(\theta_0 + \theta_1\vec{x}_1 + \cdots + \theta_n\vec{x}_n) - \vec{y}\right\|_2^2 + \lambda\left\|\theta\right\|_1$$

- Performs L_1 regularisation, i.e., adds penalty equivalent to absolute value of the magnitude of coefficients.
- Also addresses overfitting, but in addition tends to encourage coefficients to become zero rather than near-zero $\implies$ simpler models (feature selection).

[*] α in sklearn documentation.

Example — SVM, (RBF) Kernel

The Radial Basis Function (RBF) kernel has two parameters:

γ controls how far the influence of a single training example reaches.

- Low values meaning ‘far’ and high values meaning ‘close’.
- Can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.
- If too small, then model is too constrained and cannot capture the complexity or ‘shape’ of the data.

C controls the trade off of correct classification of training examples against maximisation of the decision function’s margin.

- For larger values of C, a smaller margin will be accepted if the decision function is better at classifying all training points correctly.
- A lower C will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy.
- In other words C behaves as a regularisation parameter in the SVM.

See interactive demo of effects of parameters (good but uses $1/\sigma$ instead of γ):

Support Vector Machine in Javascript

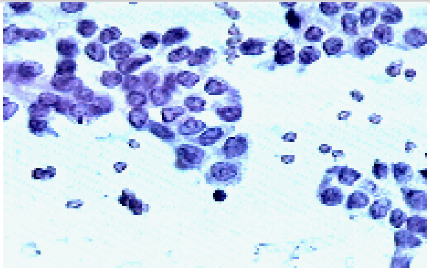
and SKLearn documentation RBF SVM Parameters

Outline

Wisconsin Dataset — Breast Cancer (WDBC)†

Outline

- Aims to predict breast cancer diagnosis based on Fine Needle Aspiration (FNA).
- Resulting classifier using on these nine features successfully diagnosed 97% of new cases.



Construction

- FNA's were done on a total of 569 patients, samples were stained to help differentiate distinguished cell nuclei
- Samples were classified as cancer-based through biopsy and historical confirmation. Non-cancer samples were confirmed by biopsy or follow ups.
- Users then chose areas of the FNA with minimal overlap between nuclei; they then took scans utilising a digital camera.
- `Xcyt` was used to create approximate boundaries, which would then used a process called snakes which converged to give the exact nuclei boundary.
- Once the boundaries for the nuclei were set, calculations were made (of mean, standard error, and max) resulting in 3×10 features.

*Machine Learning for Cancer Diagnosis and Prognosis.

WDBC — Load Data

```
UCI = "https://archive.ics.uci.edu/ml/machine-learning-databases/"
DATA_URL = f"{UCI}/breast-cancer-wisconsin/wdbc.data"
DATA_LOCAL = "data/wdbc.data"
SEED = 42

names = ['id_number', 'diagnosis', 'radius_mean',
        'texture_mean', 'perimeter_mean', 'area_mean',
        'smoothness_mean', 'compactness_mean', 'concavity_mean',
        'concave_points_mean', 'symmetry_mean',
        'fractal_dimension_mean', 'radius_se', 'texture_se',
        'perimeter_se', 'area_se', 'smoothness_se',
        'compactness_se', 'concavity_se', 'concave_points_se',
        'symmetry_se', 'fractal_dimension_se',
        'radius_worst', 'texture_worst', 'perimeter_worst',
        'area_worst', 'smoothness_worst',
        'compactness_worst', 'concavity_worst',
        'concave_points_worst', 'symmetry_worst',
        'fractal_dimension_worst']

df = pd.read_csv(DATA_URL,header=None, names=names)
df.head(10)
```

WDBC — Load Data

2

```
UCI = "https://archive.ics.uci.edu/ml/machine-learning-databases/"
DATA_URL = f"{UCI}/breast-cancer-wisconsin/wdbc.data"
DATA_LOCAL = "data/wdbc.data"
SEED = 42
```

	id_number	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave_points_mean	.
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	.
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	.
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	.
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	.
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	.
5	843786	M	12.45	15.70	82.57	477.1	0.12780	0.17000	0.15780	0.08089	.
6	844359	M	18.25	19.98	119.60	1040.0	0.09463	0.10900	0.11270	0.07400	.
7	84458202	M	13.71	20.83	90.20	577.9	0.11890	0.16450	0.09366	0.05985	.
8	844981	M	13.00	21.82	87.50	519.8	0.12730	0.19320	0.18590	0.09353	.
9	84501001	M	12.46	24.04	83.97	475.9	0.11860	0.23960	0.22730	0.08543	.

10 rows × 32 columns

```
df = pd.read_csv(DATA_URL,header=None, names=names)
df.head(10)
```

WDBC — View Data

See EDA of Breast Cancer Dataset for EDA of this dataset.

	count	mean	std	min	25%	50%	75%	max
id_number	569.0	3.037183e+07	1.250206e+08	8670.000000	869218.000000	906024.000000	8.813129e+06	9.113205e+08
radius_mean	569.0	1.412729e+01	3.524049e+00	6.981000	11.700000	13.370000	1.578000e+01	2.811000e+01
texture_mean	569.0	1.928965e+01	4.301036e+00	9.710000	16.170000	18.840000	2.180000e+01	3.928000e+01
perimeter_mean	569.0	9.196903e+01	2.429898e+01	43.790000	75.170000	86.240000	1.041000e+02	1.885000e+02
area_mean	569.0	6.548891e+02	3.519141e+02	143.500000	420.300000	551.100000	7.827000e+02	2.501000e+03
smoothness_mean	569.0	9.636028e-02	1.406413e-02	0.052630	0.086370	0.095870	1.053000e-01	1.634000e-01
compactness_mean	569.0	1.043410e-01	5.281276e-02	0.019380	0.064920	0.092630	1.304000e-01	3.454000e-01
concavity_mean	569.0	8.879932e-02	7.971981e-02	0.000000	0.029560	0.061540	1.307000e-01	4.268000e-01
concave_points_mean	569.0	4.891915e-02	3.880284e-02	0.000000	0.020310	0.033500	7.400000e-02	2.012000e-01
symmetry_mean	569.0	1.811619e-01	2.741428e-02	0.106000	0.161900	0.179200	1.957000e-01	3.040000e-01
fractal_dimension_mean	569.0	6.279761e-02	7.060363e-03	0.049960	0.057700	0.061540	6.612000e-02	9.744000e-02
radius_se	569.0	4.051721e-01	2.773127e-01	0.111500	0.232400	0.324200	4.789000e-01	2.873000e+00
texture_se	569.0	1.216853e+00	5.516484e-01	0.360200	0.833900	1.108000	1.474000e+00	4.885000e+00
perimeter_se	569.0	2.866059e+00	2.021855e+00	0.757000	1.606000	2.287000	3.357000e+00	2.198000e+01
area_se	569.0	4.033708e+01	4.549101e+01	6.802000	17.850000	24.530000	4.519000e+01	5.422000e+02
smoothness_se	569.0	7.040979e-03	3.002518e-03	0.001713	0.005169	0.006380	8.146000e-03	3.113000e-02
compactness_se	569.0	2.547814e-02	1.790818e-02	0.002252	0.013080	0.020450	3.245000e-02	1.354000e-01
concavity_se	569.0	3.189372e-02	3.018606e-02	0.000000	0.015090	0.025890	4.205000e-02	3.960000e-01
concave_points_se	569.0	1.179614e-02	6.170285e-03	0.000000	0.007638	0.010930	1.471000e-02	5.279000e-02
symmetry_se	569.0	2.054230e-02	8.266372e-03	0.007882	0.015160	0.018730	2.348000e-02	7.895000e-02
fractal_dimension_se	569.0	3.794904e-03	2.646071e-03	0.000895	0.002248	0.003187	4.558000e-03	2.984000e-02
radius_worst	569.0	1.626919e+01	4.833242e+00	7.930000	13.010000	14.970000	1.879000e+01	3.604000e+01
texture_worst	569.0	2.567722e+01	6.146258e+00	12.020000	21.080000	25.410000	2.972000e+01	4.954000e+01
perimeter_worst	569.0	1.072612e+02	3.360254e+01	50.410000	84.110000	97.660000	1.254000e+02	2.512000e+02
area_worst	569.0	8.805831e+02	5.693570e+02	185.200000	515.300000	686.500000	1.084000e+03	4.254000e+03

WDBC — View Data

See EDA of Breast Cancer Dataset for EDA of this dataset.

	count	mean	std	min	25%	50%	75%	max
id_number	569.0	3.037183e+07	1.250206e+08	8670.000000	869218.000000	906024.000000	8.813129e+06	9.113205e+08
radius_mean	569.0	1.412729e+01	3.524049e+00	6.981000	11.700000	13.370000	1.578000e+01	2.811000e+01
texture_mean	569.0	1.928965e+01	4.301036e+00	9.710000	16.170000	18.840000	2.180000e+01	3.928000e+01
perimeter_mean	569.0	9.196903e+01	2.429898e+01	43.790000	75.170000	86.240000	1.041000e+02	1.885000e+02
area_mean	569.0	6.548891e+02	3.519141e+02	143.500000	420.300000	551.100000	7.827000e+02	2.501000e+03
smoothness_mean	569.0	9.636028e-02	1.406413e-02	0.052630	0.086370	0.095870	1.053000e-01	1.634000e-01
compactness_mean	569.0	1.043410e-01	5.281276e-02	0.019380	0.064920	0.092630	1.304000e-01	3.454000e-01
concavity_mean	569.0	8.879932e-02	7.971981e-02	0.000000	0.029560	0.061540	1.307000e-01	4.268000e-01
concave_points_mean	569.0	4.891915e-02	3.880284e-02	0.000000	0.020310	0.033500	7.400000e-02	2.012000e-01
symmetry_mean	569.0	1.811619e-01	2.741428e-02	0.106000	0.161900	0.179200	1.957000e-01	3.040000e-01
fractal_dimension_mean	569.0	6.279761e-02	7.060363e-03	0.049960	0.057700	0.061540	6.612000e-02	9.744000e-02
radius_se	569.0	4.051721e-01	2.773127e-01	0.111500	0.232400	0.324200	4.789000e-01	2.873000e+00
texture_se	569.0	1.216853e+00	5.516484e-01	0.360200	0.833900	1.108000	1.474000e+00	4.885000e+00
perimeter_se	569.0	2.866059e+00	2.021855e+00	0.757000	1.606000	2.287000	3.357000e+00	2.198000e+01
area_se	569.0	4.033708e+01	4.549101e+01	6.802000	17.850000	24.530000	4.519000e+01	5.422000e+02
smoothness_se	569.0	7.040979e-03	3.002518e-03	0.001713	0.005169	0.006380	8.146000e-03	3.113000e-02
compactness_se	569.0	2.547814e-02	1.790818e-02	0.002252	0.013080	0.020450	3.245000e-02	1.354000e-01
concavity_se	569.0	3.189372e-02	3.018606e-02	0.000000	0.015090	0.025890	4.205000e-02	3.960000e-01
concave_points_se	569.0	1.179614e-02	6.170285e-03	0.000000	0.007638	0.010930	1.471000e-02	5.279000e-02
symmetry_se	569.0	2.054230e-02	8.266372e-03	0.007882	0.015160	0.018730	2.348000e-02	7.895000e-02
fractal_dimension_se	569.0	3.794904e-03	2.646071e-03	0.000895	0.002248	0.003187	4.558000e-03	2.984000e-02
radius_worst	569.0	1.626919e+01	4.833242e+00	7.930000	13.010000	14.970000	1.879000e+01	3.604000e+01
texture_worst	569.0	2.567722e+01	6.146258e+00	12.020000	21.080000	25.410000	2.972000e+01	4.954000e+01
perimeter_worst	569.0	1.072612e+02	3.360254e+01	50.410000	84.110000	97.660000	1.254000e+02	2.512000e+02
area_worst	569.0	8.805831e+02	5.693570e+02	185.200000	515.300000	686.500000	1.084000e+03	4.254000e+03

- no missing values
- big differences in mean/std $\Rightarrow$ need normalising.
- 30 numerical features, some of which expect to be correlated $\Rightarrow$ use PCA.

WDBC — Prepare Data

- Extract feature matrix and target column.

3
```
X = df.iloc[:, 2:].values
y = df.diagnosis.values
```

- Encode (categorical) target column.

  Could do our own mapping (see Churn). No big deal either way.

4
```
print (y[:20])

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)

print (le.transform(["M","B"]))
print (y[:20])
```

WDBC — Prepare Data

- Extract feature matrix and target column.

5
```
X = df.iloc[:, 2:].values
y = df.diagnosis.values
```

- Encode (categorical) target column.

  Could do our own mapping (see Churn). No big deal either way.

6
```
print (y[:20])

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)

print (le.transform(["M","B"]))
print (y[:20])
```

```
['M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M'
 'M' 'B']
[1 0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
```

WDBC — Typical Training

7

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=SEED)

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(solver='lbfgs')

from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=clf,
    X=X_train_scaled, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- Cross validation is easily parallelised — use option `n_jobs=-1` for all cores.

WDBC — Typical Training

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=SEED)

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(solver='lbfgs')

from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=clf,
    X=X_train_scaled, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- Split into train/test subsets …
- Cross validation is easily parallelised — use option n_jobs=-1 for all cores.

WDBC — Typical Training

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=SEED)

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(solver='lbfgs')

from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=clf,
    X=X_train_scaled, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- Split into train/test subsets . . . normalise . . .
- Cross validation is easily parallelised — use option `n_jobs=-1` for all cores.

WDBC — Typical Training

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=SEED)

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(solver='lbfgs')

from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=clf,
    X=X_train_scaled, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- Split into train/test subsets . . . normalise . . . and train.
- Cross validation is easily parallelised — use option `n_jobs=-1` for all cores.

WDBC — Typical Training

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=SEED)

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(solver='lbfgs')

from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=clf,
    X=X_train_scaled, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

```
CV accuracy scores: [0.97826087 0.97826087 0.97826087 0.95652174 1.
1.
 0.97777778 0.97777778 0.95555556 0.93333333]
CV accuracy: 0.974 +/- 0.019
```

- Split into train/test subsets . . . normalise . . . and train.
- Cross validation is easily parallelised — use option `n_jobs=-1` for all cores.

WDBC — Using a Pipeline I

12
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', LogisticRegression(solver='lbfgs'))
])

scores = cross_val_score(estimator=pipeline,
    X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- A pipeline is a sequence (list) of models (scaler/filters/classifiers/…) which is passed to `cross_val_score` instead of classifier as in previous slide.
- Pipes can ensure that operations (transformations, new features added) on train dataset are also applied to test/validation dataset.

WDBC — Using a Pipeline I

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', LogisticRegression(solver='lbfgs'))
])

scores = cross_val_score(estimator=pipeline,
    X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- A pipeline is a sequence (list) of models (scaler/filters/classifiers/. . .) which is passed to `cross_val_score` instead of classifier as in previous slide.
- Pipes can ensure that operations (transformations, new features added) on train dataset are also applied to test/validation dataset.

WDBC — Using a Pipeline I

14

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', LogisticRegression(solver='lbfgs'))
])

scores = cross_val_score(estimator=pipeline,
    X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

```
CV accuracy scores: [0.97826087 0.97826087 0.97826087 0.95652174 1.
1.
 0.97777778 0.97777778 0.95555556 0.93333333]
CV accuracy: 0.974 +/- 0.019
```

- A pipeline is a sequence (list) of models (scaler/filters/classifiers/. . .) which is passed to `cross_val_score` instead of classifier as in previous slide.
- Pipes can ensure that operations (transformations, new features added) on train dataset are also applied to test/validation dataset.

WDBC — Using a Pipeline II

15

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=2)),
    ('clf', LogisticRegression(solver='lbfgs'))
])

scores = cross_val_score(estimator=pipeline,
    X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- Inserting more steps into pipeline is trivial — here PCA with 2 principal components.
- OK, model is much simpler but I have lost some accuracy . . . perhaps I was too aggressive in picking 2 . . .

WDBC — Using a Pipeline II

16

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=2)),
    ('clf', LogisticRegression(solver='lbfgs'))
])

scores = cross_val_score(estimator=pipeline,
    X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

- Inserting more steps into pipeline is trivial — here PCA with 2 principal components.
- OK, model is much simpler but I have lost some accuracy . . . perhaps I was too aggressive in picking 2 . . .

WDBC — Using a Pipeline II

17

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=2)),
    ('clf', LogisticRegression(solver='lbfgs'))
])

scores = cross_val_score(estimator=pipeline,
    X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

```
CV accuracy scores: [0.93478261 0.91304348 0.97826087 0.89130435 0.97826087 0.95555556
 0.95555556 0.93333333 0.97777778 0.91111111]
CV accuracy: 0.943 +/- 0.030
```

- Inserting more steps into pipeline is trivial — here PCA with 2 principal components.
- OK, model is much simpler but I have lost some accuracy . . . perhaps I was too aggressive in picking 2 . . .

Digression . . . PCA

- What would be a good choice for the number of principal components?
- Or, more importantly, what metric could we use to determine this?

We could use PCA **specific information** that reports on the amount of variation in the components . . . to date we have used the arbitrary limit of explaining 95% of the variation . . . (but why 95%?) . . .

```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

pca = PCA(n_components=30)
X_train_scaled_pca = pca.fit_transform(X_train_scaled)
print(np.cumsum(pca.explained_variance_ratio_))
```

Digression . . . PCA

- What would be a good choice for the number of principal components?
- Or, more importantly, what metric could we use to determine this?

We could use PCA **specific information** that reports on the amount of variation in the components . . . to date we have used the arbitrary limit of explaining 95% of the variation . . . (but why 95%?) . . .

19

```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

pca = PCA(n_components=30)
X_train_scaled_pca = pca.fit_transform(X_train_scaled)
print(np.cumsum(pca.explained_variance_ratio_))
```

Digression . . . PCA

- What would be a good choice for the number of principal components?
- Or, more importantly, what metric could we use to determine this?

We could use PCA **specific information** that reports on the amount of variation in the components . . . to date we have used the arbitrary limit of explaining 95% of the variation . . . (but why 95%?) . . .

20

```python
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

pca = PCA(n_components=30)
X_train_scaled_pca = pca.fit_transform(X_train_scaled)
print(np.cumsum(pca.explained_variance_ratio_))
```

```
[0.43502782 0.63002788 0.72784307 0.79270717 0.84524094 0.88636894
 0.90872484 0.92520437 0.93900488 0.95105751 0.96162316 0.97045804
 0.97818188 0.98349877 0.986399                        16898 0.9929821
 0.99461913 0.99565834 0.996656                        4552 0.99892219
 0.99941849 0.99969478 0.99992059 0.99997136 0.99999594 1.        ]
```

$\Rightarrow$ 10 components

Digression …PCA

- What would be a good choice for the number of principal components?
- Or, more importantly, what metric could we use to determine this?

We could use PCA **specific information** that reports on the amount of variation in the components … to date we have used the arbitrary limit of explaining 95% of the variation …(but why 95%?) …

```python
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
X_train_scaled = ss.fit_transform(X_train)
X_test_scaled = ss.transform(X_test)

pca = PCA(n_components=30)
X_train_scaled_pca = pca.fit_transform(X_train_scaled)
print(np.cumsum(pca.explained_variance_ratio_))
```

```
[0.43502782 0.63002788 0.72784307 0.79270717 0.84524094 0.88636894
 0.90872484 0.92520437 0.93900488 0.95105751 0.96162316 0.97045804
 0.97818188 0.98349877 0.986399                      16898 0.9929821
 0.99461913 0.99565834 0.996656                      4552 0.99892219
 0.99941849 0.99969478 0.99992059 0.99997136 0.99999594 1. ]
```

$\Rightarrow$ 10 components

OK, this works (as with 10 components the classifier accuracy is back > 0.97 (next slide)) and is easy since we are using PCA specific metrics.
However shouldn't the overriding metric be based on how much the model accuracy is affected?

Digression . . . PCA

22

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
pipe_lr = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=10)),
    ('clf', LogisticRegression(solver='lbfgs'))
])

from sklearn.model_selection import KFold, cross_val_score
scores = cross_val_score(estimator=pipe_lr, X=X_train, y=y_train, cv=10, n_jobs=-1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),np.std(scores)))
```

```
CV accuracy scores: [1.         0.97826087 0.97826087 0.93478261 0.97826087 1.
 0.97777778 0.95555556 0.95555556 0.97777778]
CV accuracy: 0.974 +/- 0.019
```

- With 10 components the accuracy is 97.4%.

Recap of where we are

We have a dataset where:

- Clean dataset with no missing values, but have 30 numerical (continuous) features with reasonable expectation of multi-collinearity issues[‡].
- Given the dimension of 30, it seems reasonable to apply PCA, but how many principal components should we pick?

  `n_components`
- Given the suspected multi-collinearity, the regularisation in the logistic regression is important.
  - What type of penalty (L_1 vs L_2) should we use?

    `penalty`
  - How important should be the penalty be?

    `C`

We want a general procedure to determine optimal values of these hyperparameters:

- First approach is to generate **validation curves** which will look at each parameter in turn.
- Then we will look at more automatic techniques — grid and random searches.

[‡]Issues are there, ~~we~~ you could verify this by generating the correlation matrix . . .

Recap of where we are

We have a dataset where:

- Clean dataset with no missing values, but have 30 numerical (continuous) features with reasonable expectation of multi-collinearity issues[‡].
- Given the dimension of 30, it seems reasonable to apply PCA, but how many principal components should we pick?

  `n_components`
- Given the suspected multi-collinearity, the regularisation in the logistic regression is important.
  - What type of penalty (L_1 vs L_2) should we use?

    `penalty`
  - How important should be the penalty be?

    `C`

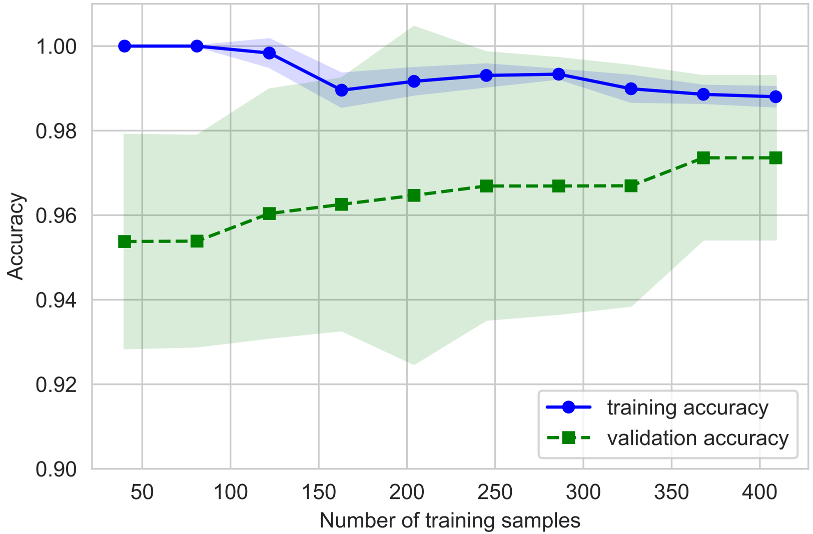
We want a general procedure to determine optimal values of these hyperparameters:

- First approach is to generate **validation curves** which will look at each parameter in turn.
- Then we will look at more automatic techniques — grid and random searches.

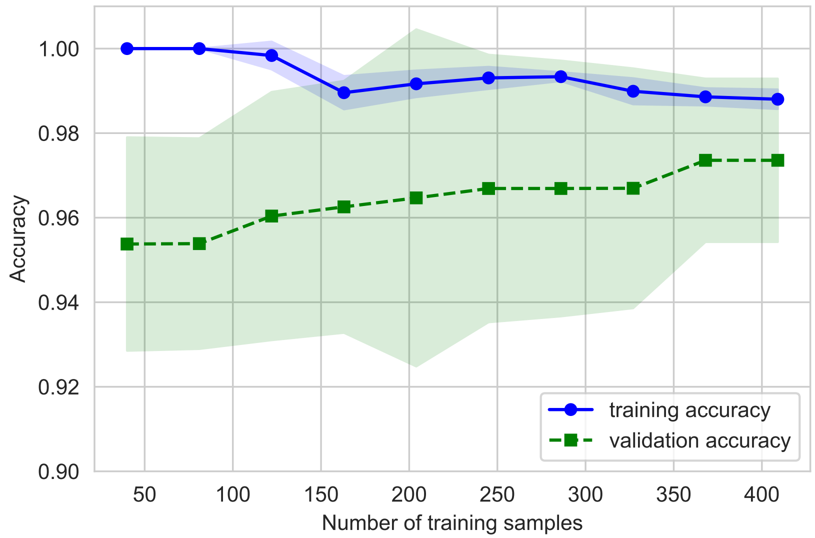
[‡]Issues are there, ~~we~~ you could verify this by generating the correlation matrix …

Outline

Learning Curve



Learning Curve



The learning curve shows how the model metrics change as the number of training samples increase (or over line).

- An under-fit model would have a flat/decreasing training accuracy.
- An over-fit model tends to have a validation accuracy that decreases to a point and begins increasing again.
- The learning curve is a tool for finding out if an estimator would benefit from more data, or if the model is too simple (under-fit/biased).

Learning Curve via Pipelines I

We pass the pipeline to the `learning_curve` function and specify values for the `train_sizes` ...

23
```
from sklearn.model_selection import learning_curve
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', LogisticRegression(solver='lbfgs'))
])
train_sizes, train_scores, test_scores = learning_curve(estimator=pipeline,
    X=X_train, y=y_train,
    train_sizes=np.linspace(0.1, 1.0, 10),
    cv=10,
    n_jobs=1)
```

Then compute statistics from the generated scores ...

24
```
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

Learning Curve via Pipelines I

We pass the pipeline to the `learning_curve` function and specify values for the `train_sizes` . . .

25
```
from sklearn.model_selection import learning_curve
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', LogisticRegression(solver='lbfgs'))
])
train_sizes, train_scores, test_scores = learning_curve(estimator=pipeline,
    X=X_train, y=y_train,
    train_sizes=np.linspace(0.1, 1.0, 10),
    cv=10,
    n_jobs=1)
```

Then compute statistics from the generated scores . . .

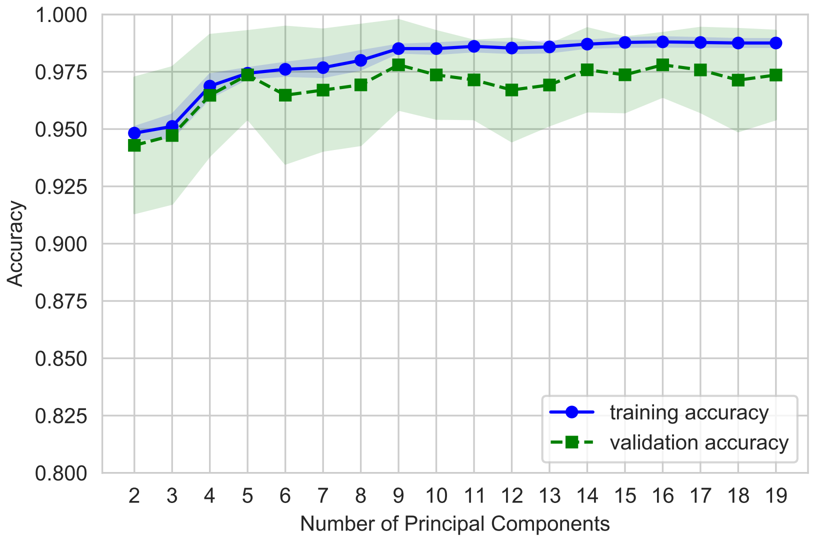
26
```
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

Learning Curve via Pipelines II

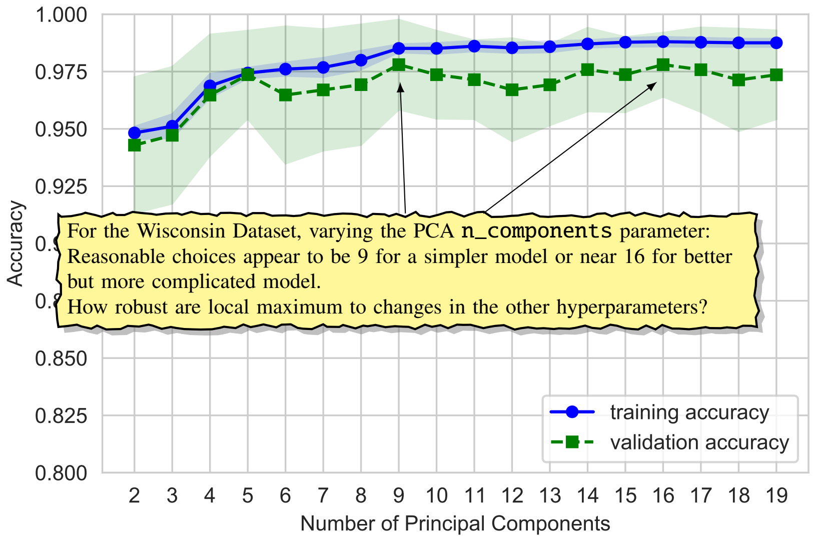
Finally, we generate the actual learning curve using the usual plot and pimping code.

```python
plt.plot(train_sizes, train_mean, color='blue', marker='o',
    markersize=5, label='training accuracy')
plt.fill_between(train_sizes, train_mean + train_std, train_mean - train_std,
    alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean, color='green', linestyle='--', marker='s',
    markersize=5, label='validation accuracy')
plt.fill_between(train_sizes, test_mean + test_std, test_mean - test_std,
    alpha=0.15, color='green')

plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim(0.9, 1.01)

plt.savefig("LC.pdf", bbox_inches="tight")
plt.show()
```

Validation Curve — PCA, `n_components`



Validation Curve — PCA, `n_components`



For the Wisconsin Dataset, varying the PCA `n_components` parameter: Reasonable choices appear to be 9 for a simpler model or near 16 for better but more complicated model.
How robust are local maximum to changes in the other hyperparameters?

The Validation Curve shows the sensitivity between model's accuracy with change in some (hyper-)parameter of the model.

- Two curves are present in a validation curve — one for the training set score and one for the cross-validation score.
- Ideally validation score and the training score look as similar as possible.
- A validation curve is used to evaluate an existing model based on hyper-parameters and is not used to tune a model. This is because, if we tune the model according to the validation score, the model may be biased towards the specific data against which the model is tuned; thereby, not being a good estimate of the generalisation of the model.

Validation Curve — PCA, `n_components` II

We pass the pipeline and (hyper-)parameter info to the `validation_curve` function, . . .

28
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=10)),
    ('clf', LogisticRegression(solver='liblinear', penalty='l2'))
])

from sklearn.model_selection import validation_curve
param_range = range(2,20)
train_scores, test_scores = validation_curve(
    estimator=pipeline, X=X_train, y=y_train, cv=10,
    param_name='pca__n_components', param_range=param_range)
```

Parameter name is concatenation of pipeline step name, `pca`, and the parameter name `n_components`.

Then compute statistics from the generated scores . . .

29
```
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

Validation Curve — PCA, `n_components` II

We pass the pipeline and (hyper-)parameter info to the `validation_curve` function, …

30
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=10)),
    ('clf', LogisticRegression(solver='liblinear', penalty='l2'))
])

from sklearn.model_selection import validation_curve
param_range = range(2,20)
train_scores, test_scores = validation_curve(
    estimator=pipeline, X=X_train, y=y_train, cv=10,
    param_name='pca__n_components', param_range=param_range)
```

Parameter name is concatenation of pipeline step name, `pca`, and the parameter name `n_components`.

Then compute statistics from the generated scores …

31
```
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

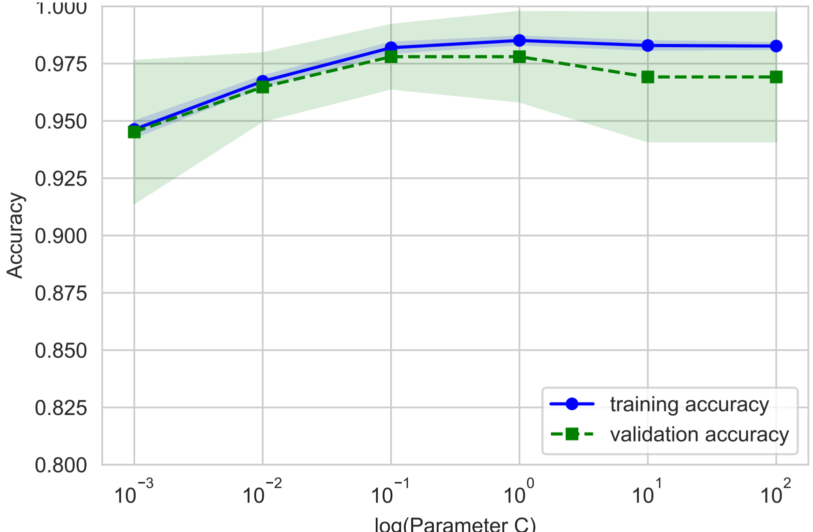
Validation Curve — PCA, `n_components` III

Finally, we generate the actual learning curve using the usual plot and pimping code.

```
plt.plot(param_range, train_mean, color='blue', marker='o', markersize=5,
    label='training accuracy')
plt.fill_between(param_range, train_mean + train_std, train_mean - train_std,
    alpha=0.15, color='blue')

plt.plot(param_range, test_mean, color='green', marker='s', markersize=5,
    linestyle='--', label='validation accuracy')
plt.fill_between(param_range, test_mean + test_std, test_mean - test_std,
    alpha=0.15, color='green')

plt.xlabel('Number of Principal Components')
plt.ylabel('Accuracy')
plt.xticks(range(2,20))
plt.legend(loc='lower right')
plt.ylim(0.8, 1.0)

plt.savefig("VC__pca__n_components.pdf", bbox_inches="tight")
plt.show()
```

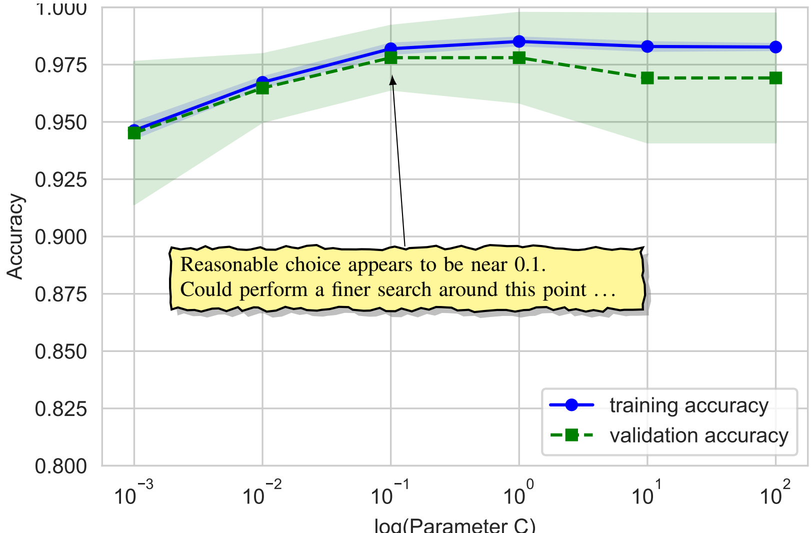
Validation Curve — LogisticRegression, c

We can also examine other hyper-parameters in our pipeline — for example, the regulerisation parameter, *C*:



Validation Curve — LogisticRegression, c I

We can also examine other hyper-parameters in our pipeline — for example, the regulerisation parameter, C:



Reasonable choice appears to be near 0.1.
Could perform a finer search around this point ...

Validation Curve — LogisticRegression, C II

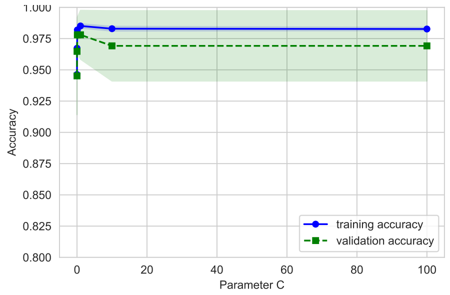
Code is near identical to that we used for modifying the n_components of the pca step:

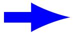
33
```
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('pca', PCA(n_components=9)),
    ('clf', LogisticRegression(solver='liblinear', penalty='l2'))
])

from sklearn.model_selection import validation_curve
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
train_scores, test_scores = validation_curve(estimator=pipeline,
    X=X_train, y=y_train, cv=10,
    param_name='clf__C', param_range=param_range)
```

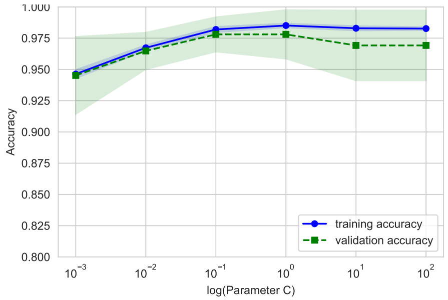
- Set number of PCA components at 9 based on results to date.
- Range of parameter C is large so prefer a geometric (constant factor, not constant difference) sweep. Could use np.logspace here.

Validation Curve — LogisticRegression, C III

Rest of code is identical (except for obvious label differences) and use of `plt.xscale('log')`







Without `plt.xscale('log')`

The interesting region is compressed and difficult to read.

With `plt.xscale('log')`

Interesting region is readable.

Recap of where we are

- Given a pipeline / model we “can” tune each of the hyperparameters by constructing a validation curve.
- Problems . . . (manual approach is not practical anything but small problems)
  - Can have a huge (100s – 1,000s) number of parameters.
  - A sequence of 1-dimensional searches is not the same as one d-dimensional search — interplay between hyper-parameters.
- Approaches / Techniques . . .
  - Grid Search — Systematic, regular, predetermined deterministic sample of parameter space.
  - Random Search — Non-adaptive, random sample of parameter space.
  - Bayesian Search — Adaptive, random sample of parameter space.