SOME EXPERIMENTS ON THE PREDICTIVE POWER OF ML MODELS

The Case of the “Shortest Path Problem”

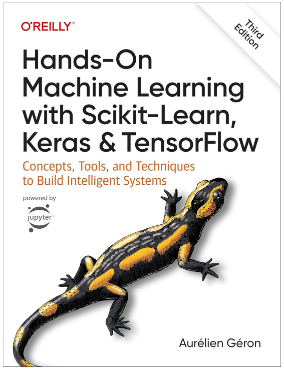
Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria

Rationale

- Interest in applying ML to aid SC with exploring “abstract search spaces”.
    - Repeatedly choose the best “next action” from a set of possible candidates.
    - Good choices may speedup the search, bad choices may slow them down.
    - But the correctness of the result does not depend on the quality of the choices.
    - Typically there is no efficient SC algorithm to make good choices.
- Start with some simple experiments.
    - Search for shortest paths in directed graphs.
    - Choice is the next node along such a path.
    - Problem can be actually solved by an efficient algorithm (Floyd–Warshall).
    - This facilitates the preparation and evaluation of experiments.
- Get familiar with ML software, methods, processes.

Not just high-level talking about ML but really “getting my hands dirty”.

Machine Learning Textbooks


O'REILLY®

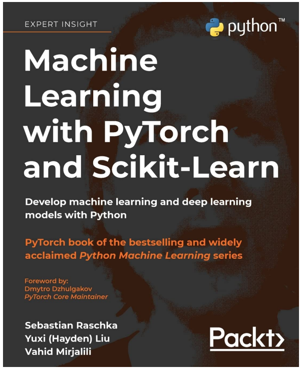
Third Edition

Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow

Concepts, Tools, and Techniques to Build Intelligent Systems

powered by jupyter

Aurélien Géron


EXPERT INSIGHT

python™

Machine Learning with PyTorch and Scikit-Learn

Develop machine learning and deep learning models with Python

PyTorch book of the bestselling and widely acclaimed *Python Machine Learning* series

Foreword by:
Dmytro Dzhulgakov
PyTorch Core Maintainer

Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili

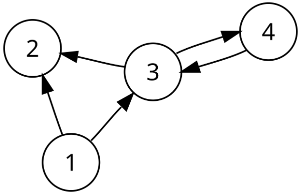
Packt>

Mostly relied on [Géron, 2022] for guidance.

The Problem

- Given: a directed graph G with n nodes and two nodes i, j.
  - $G = (V, E)$, $n \in \mathbb{N}$, $V = \mathbb{N}_n$, $E : \mathbb{N}_n \times \mathbb{N}_n \to$ Bool; $i, j \in \mathbb{N}_n$.
- Find: the (e.g., smallest) *next node* $k \in \mathbb{N}_n$ on a shortest path from i to j in G.
  - A path with minimal *length*, i.e., the minimal number of edges.
  - $k = -1$, if there is no path from i to j in G.
- Alternative: find the *length* $l \in \mathbb{N}_n$ of the shortest paths from i to j in G.
  - $l = -1$, if there is no path from i to j in G.

Related problems, but not necessarily of same "difficulty".



https://commons.wikimedia.org/wiki/File:Directed_graph_no_background.svg

$next(1, 4) = 3$, $length(1, 4) = 2$.

Data Sets

For a given node number n, problem instances are stored in CSV files

Format: $length$, $next$, i, j, $G_{0,0}$, ..., $G_{n-1,n-1}$

```
-1,-1,4,2,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1
1,3,4,3,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1
0,4,4,4,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1
```

- Data sets (with only reflexive graphs) are generated by C++ programs:
  - For each graph G, all node pairs i, j are considered.
  - Shortest paths and their lengths computed by Floyd-Warshall algorithm.
  - $n = 5$: all $n^2 \cdot 2^{(n^2-n)} \simeq 2.6 \cdot 10^7$ problem instances are enumerated.
  - $n = 10$: $n^2 \cdot 1.5 \cdot 10^6 \simeq 1.5 \cdot 10^8$ instances with random graphs are generated.
    - Randomly place additional 20 and 30 edges $\rightsquigarrow$ average outdegree 2 and 3.

Training data are randomly selected from these data sets.

Training Sets

Problem: the lengths of paths are not equally distributed in data sets.

$n = 5$

length	0	1	2	3	4	-1
$\# \cdot 10^6$	5.2	10.4	6.1	1.2	0.1	3.1

$n = 10$ (20 random edges)

length	0	1	2	3	4	5	6	7	8	9	-1
$\# \cdot 10^6$	15	30	35	22	9	3	0.7	0.1	0.02	0.0001	3.4

$n = 10$ (30 random edges)

length	0	1	2	3	4	5	6	7	8	9	-1
$\# \cdot 10^6$	15	45	56	22	4	0.6	0.08	0.008	0.0006	0.00003	6.4

- Stratification: training sets with equal portion of samples for each path length.
  - $n = 5$: 100,000 samples (more possible but not needed).
  - $n = 10$ (20 edges): 375,000 samples (path lengths ≥ 8 underrepresented).
  - $n = 10$ (30 edges): 250,000 samples (path lengths ≥ 7 underrepresented).

Substantial effort needed to represent in traing set all “input classes”.

Software

- Installation of Python 3, venv, and pip.

```
apt-get install python3 python3-venv python3-pip
```

- Setup of a virtual environment:

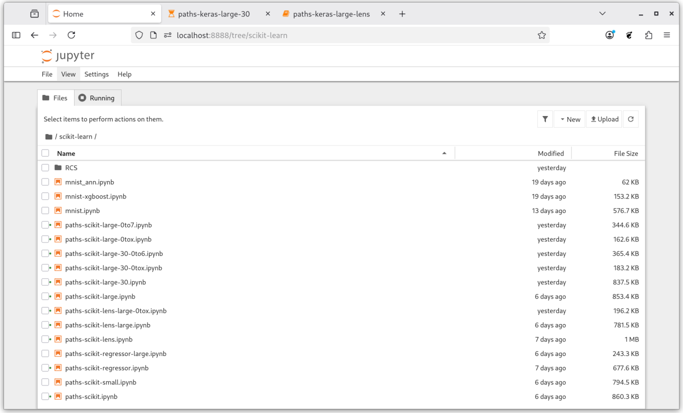
```
python3 -m venv /software/python3-ML
source /software/python3-ML/bin/activate
(python3-ML) > ...
deactivate
```

- Import of Python packages into the virtual environment:

```
pip install notebook
pip install pandas
pip install matplotlib
pip install numpy
pip install scikit-learn
pip install xgboost
pip install tensorflow
pip install tensorboard
pip install --upgrade keras-cv
pip install --upgrade keras-hub
pip install --upgrade keras
pip install keras-tuner
...
```

Jupyter Notebook Interface

```
jupyter notebook --notebook-dir=<path>
```



Select items to perform actions on them.

/ scikit-learn /

Name	Modified	File Size
RCS	yesterday	
mnist_ann.ipynb	19 days ago	62 KB
mnist-xgboost.ipynb	19 days ago	153.2 KB
mnist.ipynb	13 days ago	576.7 KB
paths-scikit-large-0to7.ipynb	yesterday	344.6 KB
paths-scikit-large-0tox.ipynb	yesterday	162.6 KB
paths-scikit-large-30-0to6.ipynb	yesterday	365.4 KB
paths-scikit-large-30-0tox.ipynb	yesterday	183.2 KB
paths-scikit-large-30.ipynb	yesterday	837.5 KB
paths-scikit-large.ipynb	6 days ago	853.4 KB
paths-scikit-lens-large-0tox.ipynb	yesterday	196.2 KB
paths-scikit-lens-large.ipynb	6 days ago	781.5 KB
paths-scikit-lens.ipynb	7 days ago	1 MB
paths-scikit-regressor-large.ipynb	6 days ago	243.3 KB
paths-scikit-regressor.ipynb	7 days ago	677.6 KB
paths-scikit-small.ipynb	6 days ago	794.5 KB
paths-scikit.ipynb	6 days ago	860.3 KB

Usually the best choice for interactive use.

Data Processing

```
import pandas as pd

train_path = "<path>.csv.gz"
train_dataframe = pd.read_csv(train_path, header=None)

X_train = train_dataframe.iloc[:,2:] # column 0 is distance (here ignored)
y_train = train_dataframe.iloc[:,1]  # column 1 is next node in path
y_train.iloc[y_train.iloc[:,] < 0] = size # "size" rather than -1 indicates "no path"

// analogous for X_valid, y_valid, X_test, y_test
...
```

Suitable for small to medium-sized training sets.

Machine Learning Software & Models

Utilize *high-level* APIs to avoid extensive Python coding.

- scikit-learn (`https://scikit-learn.org`):
  - Linear and polynomial regression, Support Vector Machines, decision trees, decision forests, multilayer perceptrons, . . .
- XGBoost (`https://xgboost.ai`):
  - Decision forests by “extreme gradient boosting”.
- Keras 3 (`https://keras.io`):
  - High-level neural network API.
  - Multiple backends: TensorFlow (Google), PyTorch (Meta AI), JAX (Google/Nvidia).

After some initial experiments, focus on two models: decision forests (in SciKit-Learn and XGBoost) and neural networks (in Keras/TensorFlow).

Regression vs Classification

- Regressor: a ML model that computes continuous values.
  - A function $r\colon \mathbb{R}^m \to \mathbb{R}^n$ for some $m, n \in \mathbb{N}_{>0}$.
- Classifier: a ML model that chooses a value from a fixed set of *classes*.
  - A function $c\colon \mathbb{R}^m \to \mathbb{N}_n$ for some $m, n \in \mathbb{N}_{>0}$.
  - May be constructed by composing a regressor r with the *softmax* function σ:

$$c([x_1, \ldots, x_m]) := \mathrm{argmax}_{k \in \mathbb{N}}\, \sigma_k(r([x_1, \ldots, x_m]))$$

$$\sigma_k([x_1, \ldots, x_n]) = \frac{\exp(x_k)}{\sum_{j=1}^{n} \exp(x_j)}$$

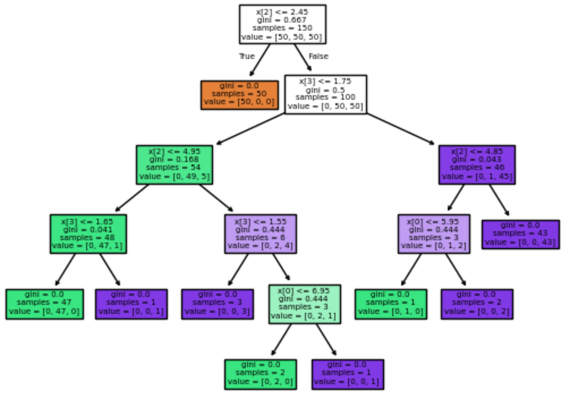
    - $\sigma_k(r([x_1, \ldots, x_m]))$: the probability that input $[x_1, \ldots, x_m]$ belongs to class k, determined from the "scores" assigned by the regressor r to each of the n classes.

ML models may be applied as regressors or as classifiers.

DECISION TREES

Decision Trees

Decision tree trained on all the iris features



x[2] <= 2.45 gini = 0.667 samples = 150 value = [50, 50, 50]
True
False
gini = 0.0 samples = 50 value = [50, 0, 0]
x[3] <= 1.75 gini = 0.5 samples = 100 value = [0, 50, 50]
x[2] <= 4.95 gini = 0.168 samples = 54 value = [0, 49, 5]
x[2] <= 4.85 gini = 0.043 samples = 46 value = [0, 1, 45]
x[3] <= 1.65 gini = 0.041 samples = 48 value = [0, 47, 1]
x[3] <= 1.55 gini = 0.444 samples = 6 value = [0, 2, 4]
x[0] <= 5.95 gini = 0.444 samples = 3 value = [0, 1, 2]
gini = 0.0 samples = 43 value = [0, 0, 43]
gini = 0.0 samples = 47 value = [0, 47, 0]
gini = 0.0 samples = 1 value = [0, 0, 1]
gini = 0.0 samples = 3 value = [0, 0, 3]
x[0] <= 6.95 gini = 0.444 samples = 3 value = [0, 2, 1]
gini = 0.0 samples = 1 value = [0, 1, 0]
gini = 0.0 samples = 2 value = [0, 0, 2]
gini = 0.0 samples = 2 value = [0, 2, 0]
gini = 0.0 samples = 1 value = [0, 0, 1]

2

https://scikit-learn.org/stable/modules/tree.html

Training: the tree is “grown” by the CART (Classification and Regression Tree) algorithm: the training set is recursively split by that decision $feature \leq threshold$ that minimizes the “Gini impurity” of the subsets weighted by their size.

Decision Forests

A single decision tree does not represent a very good predictor.

- Ensembles: combinations of multiple weak predictors.
  - The aggregated predication may be much better than each individual one.
- Random Forests: multiple decision trees are grown (independently) from random subsets of the training data.
  - Additionally, the best feature is chosen from a random subet of features.
  - Extra-Trees (extremely randomized trees): also thresholds are chosen randomly.
  - Aggregation: the prediction with the highest count wins (hard voting) or the prediction with the highest average probability wins (soft voting).
- Gradient Boosting: decision trees are constructed one after another.
  - Each decision tree is trained on the residual error of its predecessor.
  - Aggregation: the prediction is the sum of the individual ones.

Decision Forests in scikit-learn

```
from sklearn.ensemble import (ExtraTreesClassifier,
  GradientBoostingClassifier, HistGradientBoostingClassifier)
from xgboost import XGBClassifier, plot_importance, plot_tree

from sklearn.model_selection import (learning_curve, validation_curve,
  cross_val_score, cross_val_predict, LearningCurveDisplay)
from sklearn.metrics import ConfusionMatrixDisplay

import matplotlib.pyplot as plt
import numpy as np
```

For determining the next nodes, we use the classifier variants of the models.

Model Fitting and Predicting

We predict next nodes of shortest paths in graphs with $n = 5$ nodes.

```
model = XGBClassifier(random_state=42)
// or: ExtraTreesClassifier, GradientBoostingClassifier, HistGradientBoostingClassifier

model.fit(X_train, y_train)
y_pred = model.predict(X_test[0:20])

print(y_pred)
print(y_test[0:20].values)
print(1-sum([0 if elem == 0 else 1 for elem in model.predict(X_test)-y_test])/len(X_test))

[1 4 4 2 0 2 0 4 2 2 1 3 1 0 5 1 2 1 3 1]
[1 4 4 2 0 2 0 4 2 2 1 3 1 0 5 1 2 1 3 1]
0.9958
```

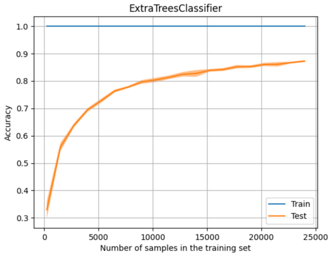
After fitting the model to the training set, it may perform predictions on the test set.

Learning Curves

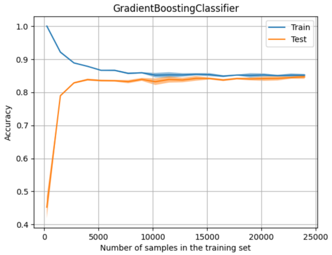
```
_ , ax = plt.subplots()
ax.set_title("ExtraTreesClassifier")
ax.grid()

LearningCurveDisplay.from_estimator(
    model, X_train, y_train, train_sizes=np.linspace(0.01,1.0,20), cv=5,
    scoring="accuracy", n_jobs=-1, ax=ax)

plt.show()
```

- Cross-Validation: the training set is split int $cv = 5$ pieces; cv copies of the model are trained, each using $cv - 1$ pieces for training and one for validation.
- Learning Curve: we repeatedly apply cross-validation for growing fractions of the training set and plot the average validation accuracy.

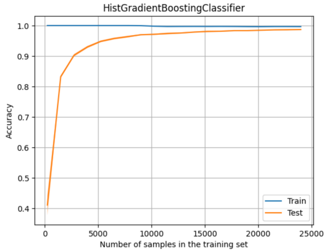
Learning Curves



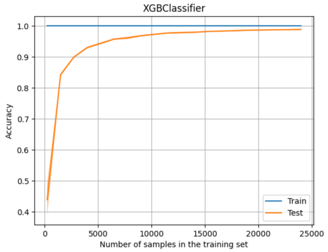
```
CPU times: user 411 ms, sys: 63.5 ms, total: 474 ms
Wall time: 43 s
```



```
CPU times: user 555 ms, sys: 80.8 ms, total: 636 ms
Wall time: 1min 56s
```



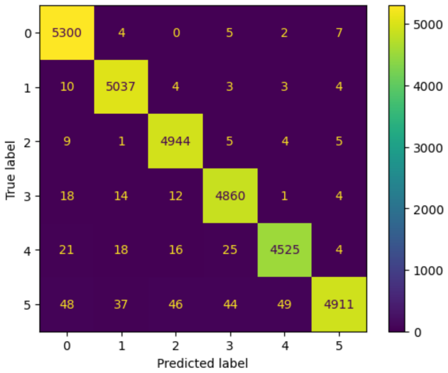
```
CPU times: user 431 ms, sys: 98 ms, total: 529 ms
Wall time: 1min 8s
```



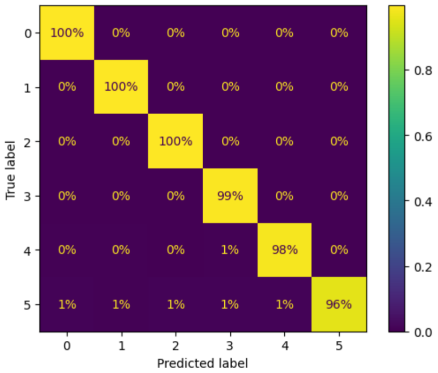
```
CPU times: user 285 ms, sys: 55.2 ms, total: 340 ms
Wall time: 25.4 s
```

Confusion Matrices

```
y_train_pred = cross_val_predict(model, X_train, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
  normalize="true", values_format=".0%")
plt.show()
```



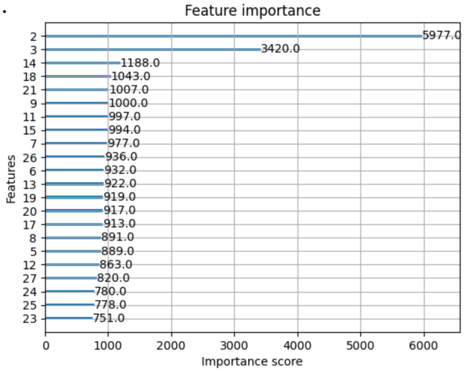
	0	1	2	3	4	5
0	5300	4	0	5	2	7
1	10	5037	4	3	3	4
2	9	1	4944	5	4	5
3	18	14	12	4860	1	4
4	21	18	16	25	4525	4
5	48	37	46	44	49	4911

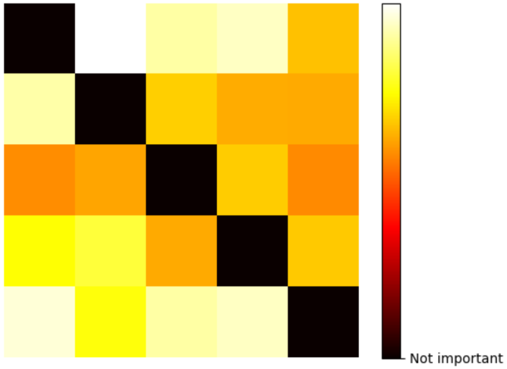


	0	1	2	3	4	5
0	100%	0%	0%	0%	0%	0%
1	0%	100%	0%	0%	0%	0%
2	0%	0%	100%	0%	0%	0%
3	0%	0%	0%	99%	0%	0%
4	0%	0%	0%	1%	98%	0%
5	1%	1%	1%	1%	1%	96%

Feature Importance

```
model = XGBClassifier(random_state=42)
model.fit(X_train, y_train)
plot_importance(model)
plt.show()
..
```

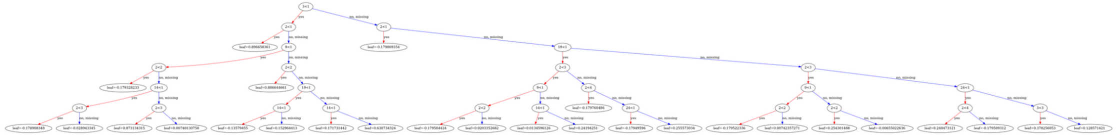




The node indices i, j are most important, the diagonal $g_{k,k}$ is ignored.

One Tree in the Forest

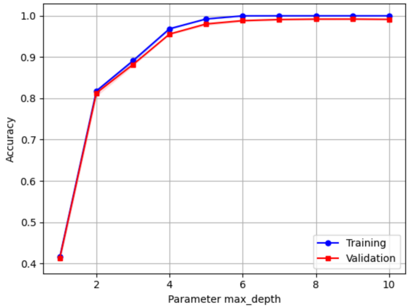
```
_, ax = plt.subplots(figsize=(60, 40))
plot_tree(model,ax=ax)
plt.show()
```



By default, there are up to 100 decision trees with maximum depth 6.

Validation Curves

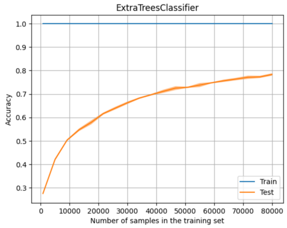
```
params=range(1,11)
train_scores, valid_scores = validation_curve(model, X_train, y_train,
    param_name="max_depth", param_range=params, scoring="accuracy", cv=5, n_jobs=-1)
train_mean = train_scores.mean(axis=1)
valid_mean = valid_scores.mean(axis=1)
plt.plot(params, train_mean, color="blue", marker="o", markersize=5, label="Training")
plt.plot(params, valid_mean, color="red", marker="s", markersize=5, label="Validation")
...
plt.show()
```



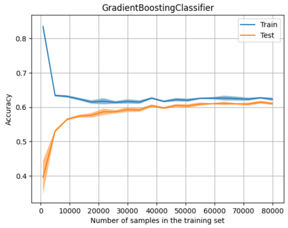
```
CPU times: user 221 ms, sys: 22.2 ms, total: 244 ms
Wall time: 22.2 s
```

Maximum depth 6 is fine (and so is the maximum number of trees).

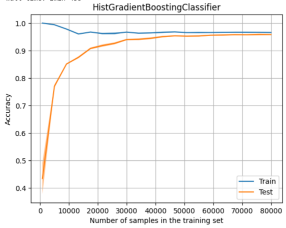
Predicting the Lengths of Shortest Paths by Classification



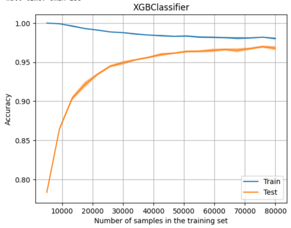
```
CPU times: user 785 ms, sys: 2.54 s, total: 3.33 s
Wall time: 2min 45s
```



```
CPU times: user 1.95 s, sys: 1.18 s, total: 3.13 s
Wall time: 8min 23s
```



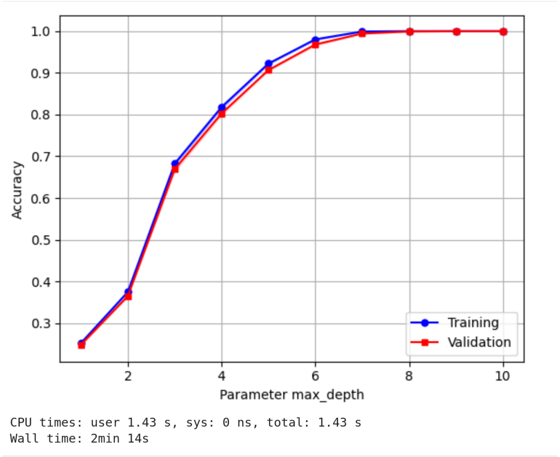
```
CPU times: user 1.09 s, sys: 425 ms, total: 1.52 s
Wall time: 3min 8s
```



```
CPU times: user 1.05 s, sys: 350 ms, total: 1.4 s
Wall time: 1min 58s
```

Please note the tripled size of the training set.

Predicting the Lengths of Shortest Paths by Classification

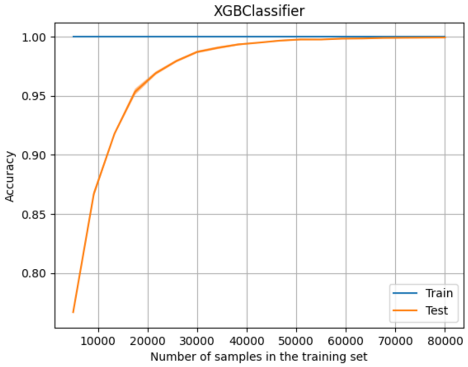


```
CPU times: user 1.43 s, sys: 0 ns, total: 1.43 s
Wall time: 2min 14s
```

Better increase the maximum depth of the decision trees to 8.

Predicting the Lengths of Shortest Paths by Classification

```
model = XGBClassifier(random_state=42, max_depth=8)
```



```
CPU times: user 1.03 s, sys: 229 ms, total: 1.26 s
Wall time: 2min 3s
```

So with deeper decision trees also very high accuracy can be achieved.

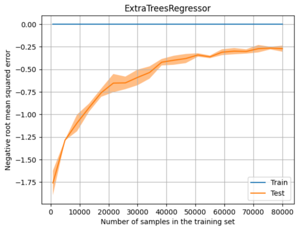
Predicting the Lengths of Shortest Paths by Regression

```
from sklearn.ensemble import (ExtraTreesRegressor, GradientBoostingRegressor,
  HistGradientBoostingRegressor)
from xgboost import XGBRegressor

model = XGBRegressor(random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test[0:10])

print(np.round(y_pred, 1))
print(y_test[0:10].values)
print(math.sqrt(sum([elem*elem for elem in model.predict(X_test)-y_test])/len(X_test))) # RMSE
print(1-sum([0 if elem == 0 else 1 for elem in np.round(model.predict(X_test))-y_test])/len(X_test))
```

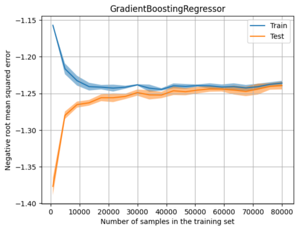
```
[ 2.5  3.2  1.1  3.8  3.8  0.8  3.9 -0.1  4.9  2.5]
[3 3 0 5 4 1 4 0 5 2]
0.521919080843475
0.6949000000000001
```

The predictions are continuous values.

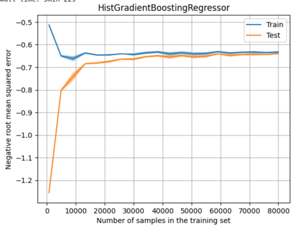
Predicting the Lengths of Shortest Paths by Regression



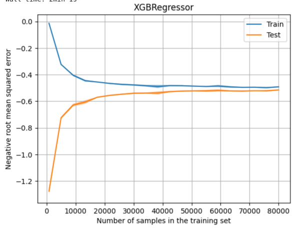
```
CPU times: user 1.97 s, sys: 359 ms, total: 2.33 s
Wall time: 5min 22s
```



```
CPU times: user 1.07 s, sys: 400 ms, total: 1.47 s
Wall time: 2min 1s
```



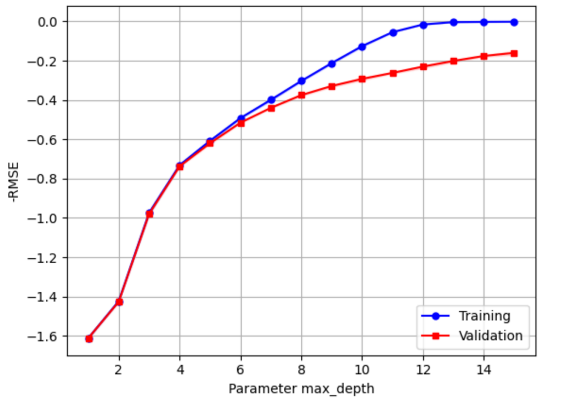
```
CPU times: user 484 ms, sys: 164 ms, total: 648 ms
Wall time: 27.2 s
```



```
CPU times: user 482 ms, sys: 67.7 ms, total: 549 ms
Wall time: 12.7 s
```

The learning curves depict the root mean square error (RMSE).

Predicting the Lengths of Shortest Paths by Regression

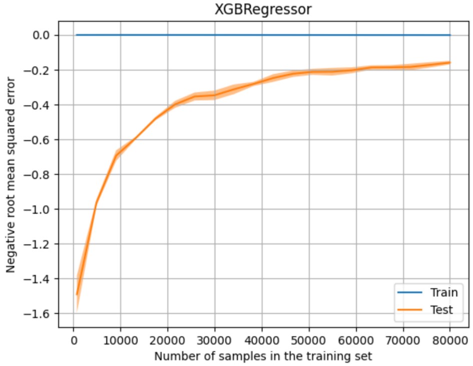


```
CPU times: user 707 ms, sys: 133 ms, total: 840 ms
Wall time: 45.9 s
```

Better increase the maximum depth of the decision trees to 15.

Predicting the Lengths of Shortest Paths by Regression

```
model = XGBRegressor(random_state=42, max_depth=15)
```

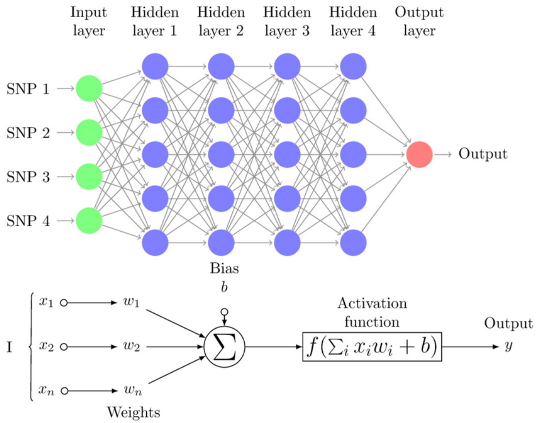


```
CPU times: user 826 ms, sys: 123 ms, total: 949 ms
Wall time: 59.1 s
```

So with much deeper decision trees $\mathrm{RMSE} \ll 0.5$ can be achieved; compared to classification, by regression the computation time is halved.

NEURAL NETWORKS

Neural Networks



http://dx.doi.org/10.3390/genes10070553

The “multilayer perceptron” (MLP) (also called: “feed-forward neural network”).

Neural Network Classifiers in Keras/TensorFlow

```
import tensorflow as tf
from tensorflow import keras

normalization = keras.layers.Normalization()
normalization.adapt(X_train.to_numpy())

def deep_net(width,depth):
    model = keras.Sequential()
    model.add(keras.layers.Input(shape=(2+size*size,)))
    model.add(normalization)                         # don’t forget to normalize the input features!
    for _ in range(depth):
        model.add(keras.layers.Dense(
            width,
            activation="selu",                       # activation function: SELU
            kernel_initializer="lecun_normal"  # kernel initializer: LeCun
        ))
    model.add(keras.layers.Dense(size+1, activation="softmax"))
    return model
```

Input layer of size $2 + n^2$, $depth$ hidden layers of size $width$, outp. layer of size $n + 1$.

Model Fitting

```
model = deep_net(2+2*size*size, 3)

# Nesterov accelerated gradient (NAG) optimizer
nag = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
model.compile(loss="sparse_categorical_crossentropy", optimizer=nag, metrics=["accuracy"])

lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.1, patience=5)
early_stopping = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)
tensorboard = tf.keras.callbacks.TensorBoard(tensorboard_logdir())

model.fit(
    X_train, y_train, validation_data = (X_valid, y_valid),
    epochs = 200,
    callbacks=[lr_scheduler, early_stopping, tensorboard])
```

At most 200 iterations over training set (“epochs”); if validation loss is not decreased for 5 epochs, learning rate is divided by 10; if validation rate is not decreased for 10 epochs, training stops; progress after each epoch is logged for TensorBoard visualization.

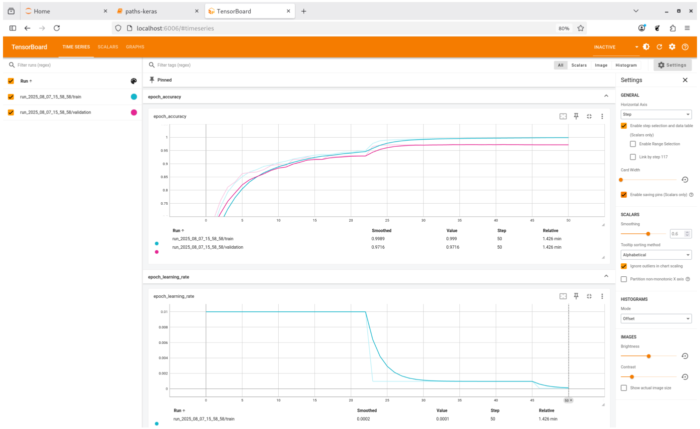
Model Fitting

```
Epoch 1/200
938/938 ------- 2s 2ms/step - accuracy: 0.4469 - loss: 1.4494 - val_accuracy: 0.6289 - val_loss: 1.0444 - learning_rate: 0.0100
Epoch 2/200
938/938 ------- 2s 2ms/step - accuracy: 0.6542 - loss: 0.9741 - val_accuracy: 0.7167 - val_loss: 0.7862 - learning_rate: 0.0100
Epoch 3/200
938/938 ------- 2s 2ms/step - accuracy: 0.7306 - loss: 0.7400 - val_accuracy: 0.7631 - val_loss: 0.6545 - learning_rate: 0.0100
Epoch 4/200
938/938 ------- 2s 2ms/step - accuracy: 0.7946 - loss: 0.5707 - val_accuracy: 0.8095 - val_loss: 0.5329 - learning_rate: 0.0100
...
Epoch 35/200
938/938 ------- 2s 2ms/step - accuracy: 0.9971 - loss: 0.0136 - val_accuracy: 0.9749 - val_loss: 0.0693 - learning_rate: 1.0000e-03
...
Epoch 42/200
938/938 ------- 2s 2ms/step - accuracy: 0.9991 - loss: 0.0098 - val_accuracy: 0.9750 - val_loss: 0.0699 - learning_rate: 1.0000e-04
Epoch 43/200
938/938 ------- 2s 2ms/step - accuracy: 0.9990 - loss: 0.0092 - val_accuracy: 0.9750 - val_loss: 0.0699 - learning_rate: 1.0000e-04
Epoch 44/200
938/938 ------- 2s 2ms/step - accuracy: 0.9993 - loss: 0.0088 - val_accuracy: 0.9748 - val_loss: 0.0698 - learning_rate: 1.0000e-04
Epoch 45/200
938/938 ------- 2s 2ms/step - accuracy: 0.9994 - loss: 0.0086 - val_accuracy: 0.9747 - val_loss: 0.0699 - learning_rate: 1.0000e-04
```

In each epoch, the training set is randomly partitioned into “mini-batches” of size 32; for each, a gradient is computed and a gradient descent step is performed; finally, the validation loss is determined.

TensorBoard

```
tensorboard --logdir tensorboard
```



	Run ↑
✓	run_2025_08_07_15_58_58/train
✓	run_2025_08_07_15_58_58/validation

epoch_accuracy

	Run ↑	Smoothed	Value	Step	Relative
●	run_2025_08_07_15_58_58/train	0.9989	0.999	50	1.426 min
●	run_2025_08_07_15_58_58/validation	0.9716	0.9716	50	1.426 min

epoch_learning_rate

	Run ↑	Smoothed	Value	Step	Relative
●	run_2025_08_07_15_58_58/train	0.0002	0.0001	50	1.426 min

Settings

GENERAL
Horizontal Axis: Step
[✓] Enable step selection and data table (Scalars only)
[ ] Enable Range Selection
[ ] Link by step 117
Card Width
[✓] Enable saving pins (Scalars only)

SCALARS
Smoothing: 0.6
Tooltip sorting method: Alphabetical
[✓] Ignore outliers in chart scaling
[ ] Partition non-monotonic X axis

HISTOGRAMS
Mode: Offset

IMAGES
Brightness
Contrast
[ ] Show actual image size

The training progress is captured and can be visualized.

Model Predicting

```
y_pred = model.predict(X_test[0:5])

print(y_pred.round(2))
print(y_pred.argmax(axis=-1))
print(y_test[0:5].values)
print(1-sum([0 if d == 0 else 1 for d in model.predict(X_test).argmax(axis=-1)-y_test])/len(X_test))
```

```
[[0.   0.02 0.02 0.15 0.   0.81]
 [1.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.  ]
 [0.   0.   0.   1.   0.   0.  ]
 [0.   0.   1.   0.   0.   0.  ]]
[5 0 1 3 2]
[5 0 1 3 2]
0.9741
```

The predictions of the model are class *probabilities*.

Hyperparameter Tuning

```
import keras_tuner as kt

def build_model(hp):
    width = hp.Int("width", min_value=2+size*size, max_value=2+5*size*size, step=size*size)
    depth = hp.Int("depth", min_value=1, max_value=5)
    model = deep_net(width, depth)
    nag = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
    model.compile(loss="sparse_categorical_crossentropy", optimizer=nag, metrics=["accuracy"])
    return model

tuner = kt.GridSearch(build_model, objective="val_accuracy", overwrite=True)
tuner.search(
    X_train, y_train, validation_data = (X_valid, y_valid),
    epochs = 200,
    callbacks=[ early_stopping, lr_scheduler])
print(tuner.get_best_hyperparameters()[0].values)
best_model = tuner.get_best_models()[0]
```

Automated exploration of the hyperparameter value search space.

Neural Network Regressors in Keras/TensorFlow

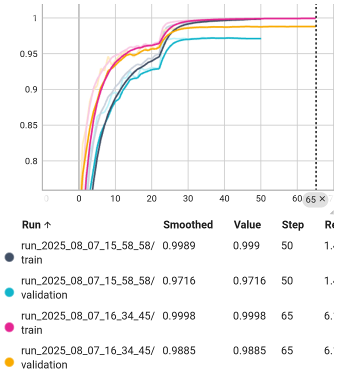
```
def deep_net(width,depth):
    model = keras.Sequential()
    model.add(keras.layers.Input(shape=(2+size*size,)))
    model.add(normalization)
    for _ in range(depth):
        model.add(keras.layers.Dense(
            width,
            activation="selu",
            kernel_initializer="lecun_normal"
        ))
    model.add(keras.layers.Dense(1)) # single neuron without activation
    return model

model = deep_net(2+2*size*size, 3)
nag = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
model.compile(loss="mse", optimizer=nag, metrics=["RootMeanSquaredError"])
```

(Root) mean square error as loss function and metrics.

Predicting Shortest Paths and Their Lengths

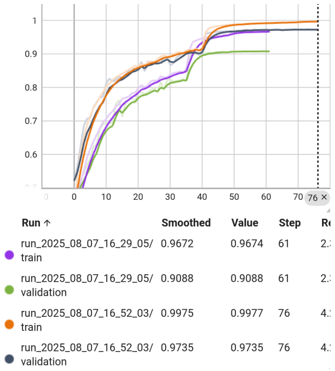
Width $2 + 2 \cdot 5^2$, depth 3, training set sizes $30{,}000$ and $60{,}000$.

next nodes (accuracy)



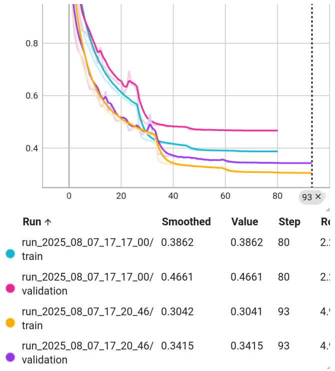
	Run ↑	Smoothed	Value	Step	R
●	run_2025_08_07_15_58_58/ train	0.9989	0.999	50	1.
●	run_2025_08_07_15_58_58/ validation	0.9716	0.9716	50	1.
●	run_2025_08_07_16_34_45/ train	0.9998	0.9998	65	6.
●	run_2025_08_07_16_34_45/ validation	0.9885	0.9885	65	6.

lengths (accuracy)



	Run ↑	Smoothed	Value	Step	R
●	run_2025_08_07_16_29_05/ train	0.9672	0.9674	61	2.
●	run_2025_08_07_16_29_05/ validation	0.9088	0.9088	61	2.
●	run_2025_08_07_16_52_03/ train	0.9975	0.9977	76	4.
●	run_2025_08_07_16_52_03/ validation	0.9735	0.9735	76	4.

lengths (RMSE)



	Run ↑	Smoothed	Value	Step	R
●	run_2025_08_07_17_17_00/ train	0.3862	0.3862	80	2.
●	run_2025_08_07_17_17_00/ validation	0.4661	0.4661	80	2.
●	run_2025_08_07_17_20_46/ train	0.3042	0.3041	93	4.
●	run_2025_08_07_17_20_46/ validation	0.3415	0.3415	93	4.

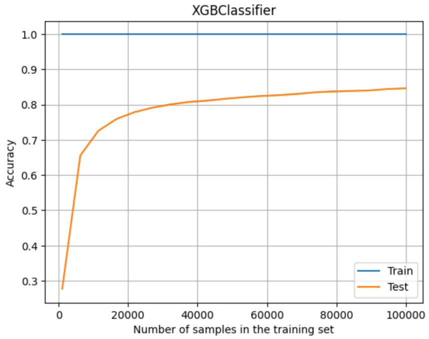
Similar accuracy/RMSE as with XGBoost, but *much* longer training times.

THE LARGER PROBLEM

Predicting the Next Nodes in Shortest Paths

Graphs with 10 nodes and 20 random edges.

```
model = XGBClassifier(random_state=42, max_depth=12)
```



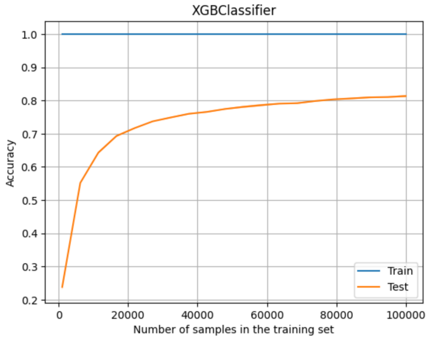
```
CPU times: user 5.09 s, sys: 1.11 s, total: 6.2 s
Wall time: 11min 58s
```

Increased training set size and maximum tree depth.

Predicting the Next Nodes in Shortest Paths

Graphs with 10 nodes and 30 random edges.

```
model = XGBClassifier(random_state=42, max_depth=12)
```

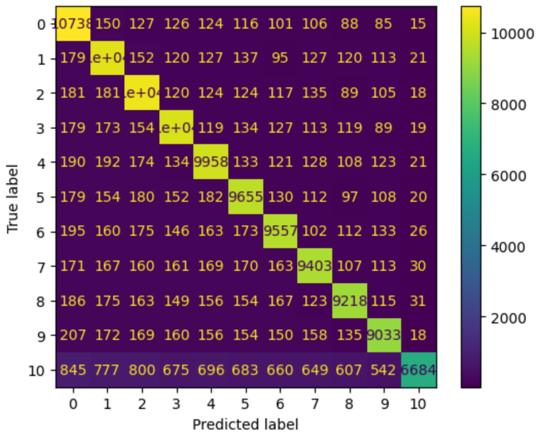


```
CPU times: user 2.99 s, sys: 33.9 s, total: 36.9 s
Wall time: 20min 35s
```

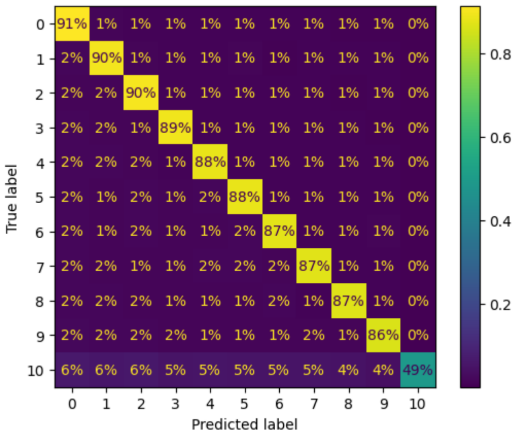
Accuracy is moderately decreased.

Predicting the Next Nodes in Shortest Paths

Graphs with 10 nodes and 20 random edges, training set size $100{,}000$.



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10
0	10738	150	127	126	124	116	101	106	88	85	15
1	179	e+0	152	120	127	137	95	127	120	113	21
2	181	181	e+0	120	124	124	117	135	89	105	18
3	179	173	154	e+0	119	134	127	113	119	89	19
4	190	192	174	134	9958	133	121	128	108	123	21
5	179	154	180	152	182	9655	130	112	97	108	20
6	195	160	175	146	163	173	9557	102	112	133	26
7	171	167	160	161	169	170	163	9403	107	113	30
8	186	175	163	149	156	154	167	123	9218	115	31
9	207	172	169	160	156	154	150	158	135	9033	18
10	845	777	800	675	696	683	660	649	607	542	6684



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10
0	91%	1%	1%	1%	1%	1%	1%	1%	1%	1%	0%
1	2%	90%	1%	1%	1%	1%	1%	1%	1%	1%	0%
2	2%	2%	90%	1%	1%	1%	1%	1%	1%	1%	0%
3	2%	2%	1%	89%	1%	1%	1%	1%	1%	1%	0%
4	2%	2%	2%	1%	88%	1%	1%	1%	1%	1%	0%
5	2%	1%	2%	1%	2%	88%	1%	1%	1%	1%	0%
6	2%	1%	2%	1%	1%	2%	87%	1%	1%	1%	0%
7	2%	2%	1%	1%	2%	2%	2%	87%	1%	1%	0%
8	2%	2%	2%	1%	1%	1%	2%	1%	87%	1%	0%
9	2%	2%	2%	2%	1%	1%	1%	2%	1%	86%	0%
10	6%	6%	6%	5%	5%	5%	5%	5%	4%	4%	49%

The predictions are now less than perfect.

Prediction Accuracy vs Path Lengths

We test the accuracy of predictions for fixed path lengths.

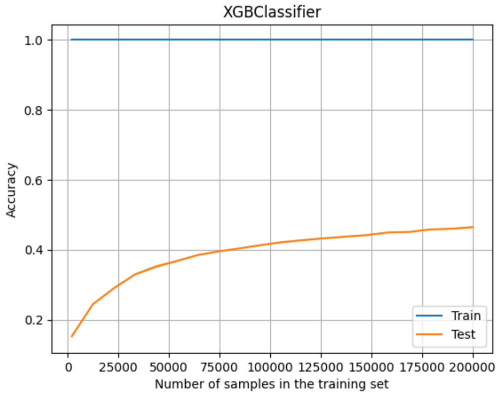
```
model.fit(X_train, y_train)
print("0:", 1-sum([0.0 if item == 0 else 1.0 for item in model.predict(X_test0)-y_test0])/len(X_test0))
...
print("10:", 1-sum([0.0 if item == 0 else 1.0 for item in model.predict(X_testx)-y_testx])/len(X_testx))
```

```
0: 1.0
1: 0.9991
2: 0.75592
3: 0.74846
4: 0.83614
5: 0.89624
6: 0.93126
7: 0.96278
8: 0.9872
9: 0.998
10: 0.5066200000000001
```

Interestingly, the predictions are less accurate for the more frequent path lengths; furthermore, the model can hardly predict the non-existence of paths.

Predicting the Lengths of Shortest Paths by Classification

```
model = XGBClassifier(random_state=42, max_depth=12, n_estimators=200)
```

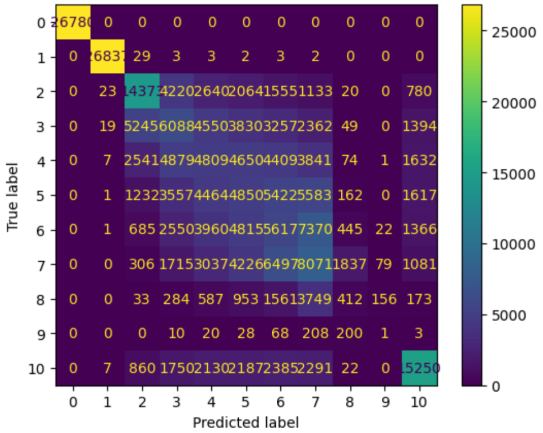


```
CPU times: user 27.6 s, sys: 7.35 s, total: 35 s
Wall time: 1h 24min 37s
```

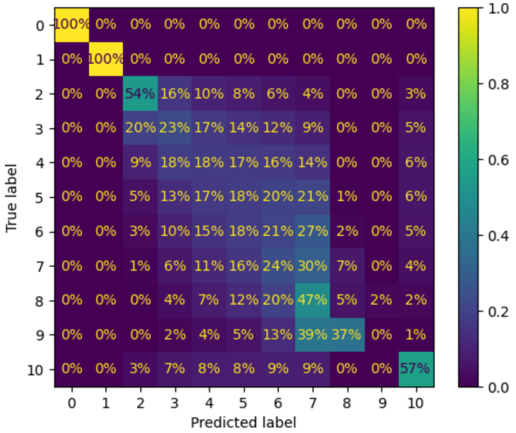
Poor accuracy, even with larger number of estimators.

Predicting the Lengths of Shortest Paths by Classification

Graphs with 10 nodes and 20 random edges, training set size $100{,}000$.



	0	1	2	3	4	5	6	7	8	9	10
0	26780	0	0	0	0	0	0	0	0	0	0
1	0	26837	29	3	3	2	3	2	0	0	0
2	0	23	14373	4220	2640	2064	1555	1133	20	0	780
3	0	19	5245	6088	4550	3830	3257	2362	49	0	1394
4	0	7	2541	4879	4809	4650	4409	3841	74	1	1632
5	0	1	1232	3557	4464	4850	5422	5583	162	0	1617
6	0	1	685	2550	3960	4815	5617	7370	445	22	1366
7	0	0	306	1715	3037	4226	6497	8071	1837	79	1081
8	0	0	33	284	587	953	1561	3749	412	156	173
9	0	0	0	10	20	28	68	208	200	1	3
10	0	7	860	1750	2130	2187	2385	2291	22	0	15250

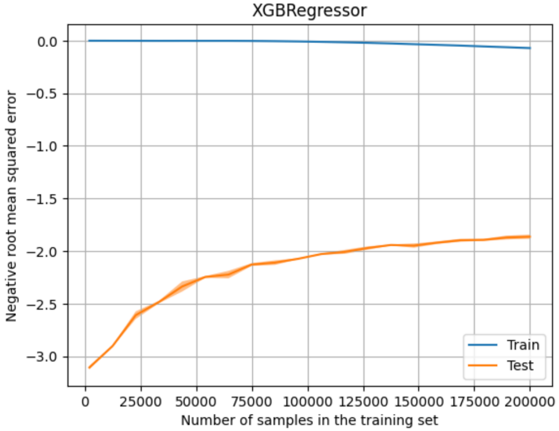


	0	1	2	3	4	5	6	7	8	9	10
0	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
2	0%	0%	54%	16%	10%	8%	6%	4%	0%	0%	3%
3	0%	0%	20%	23%	17%	14%	12%	9%	0%	0%	5%
4	0%	0%	9%	18%	18%	17%	16%	14%	0%	0%	6%
5	0%	0%	5%	13%	17%	18%	20%	21%	1%	0%	6%
6	0%	0%	3%	10%	15%	18%	21%	27%	2%	0%	5%
7	0%	0%	1%	6%	11%	16%	24%	30%	7%	0%	4%
8	0%	0%	0%	4%	7%	12%	20%	47%	5%	2%	2%
9	0%	0%	0%	2%	4%	5%	13%	39%	37%	0%	1%
10	0%	0%	3%	7%	8%	8%	9%	9%	0%	0%	57%

The predictions for path lengths greater than 2 are pretty random.

Predicting the Lengths of Shortest Paths by Regression

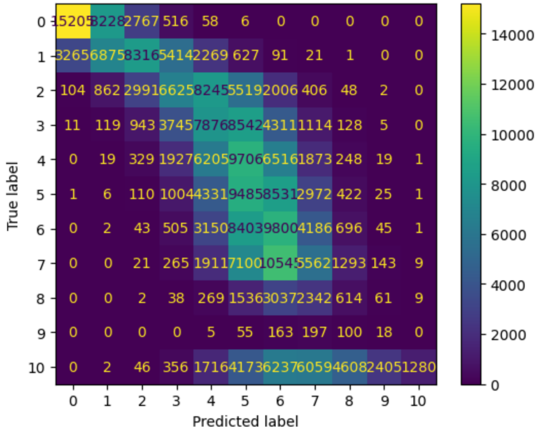
```
model = XGBRegressor(random_state=42, max_depth=10)
```



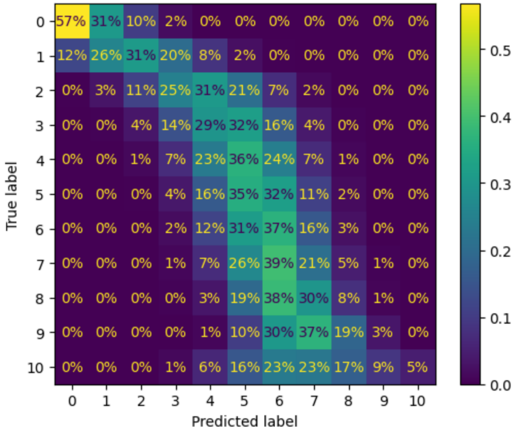
Much faster, but even with doubled training set size still a high error.

Predicting the Lengths of Shortest Paths by Regression

Graphs with 10 nodes and 20 random edges, training set size $200{,}000$.



	0	1	2	3	4	5	6	7	8	9	10
0	15205	8228	2767	516	58	6	0	0	0	0	0
1	3265	6875	8316	5414	2269	627	91	21	1	0	0
2	104	862	2991	6625	8245	5519	2006	406	48	2	0
3	11	119	943	3745	7876	8542	4311	1114	128	5	0
4	0	19	329	1927	6205	9706	6516	1873	248	19	1
5	1	6	110	1004	4331	9485	8531	2972	422	25	1
6	0	2	43	505	3150	8403	9800	4186	696	45	1
7	0	0	21	265	1911	7100	10545	5562	1293	143	9
8	0	0	2	38	269	1536	3037	2342	614	61	9
9	0	0	0	0	5	55	163	197	100	18	0
10	0	2	46	356	1716	4173	6237	6059	4608	2405	1280



	0	1	2	3	4	5	6	7	8	9	10
0	57%	31%	10%	2%	0%	0%	0%	0%	0%	0%	0%
1	12%	26%	31%	20%	8%	2%	0%	0%	0%	0%	0%
2	0%	3%	11%	25%	31%	21%	7%	2%	0%	0%	0%
3	0%	0%	4%	14%	29%	32%	16%	4%	0%	0%	0%
4	0%	0%	1%	7%	23%	36%	24%	7%	1%	0%	0%
5	0%	0%	0%	4%	16%	35%	32%	11%	2%	0%	0%
6	0%	0%	0%	2%	12%	31%	37%	16%	3%	0%	0%
7	0%	0%	0%	1%	7%	26%	39%	21%	5%	1%	0%
8	0%	0%	0%	0%	3%	19%	38%	30%	8%	1%	0%
9	0%	0%	0%	0%	1%	10%	30%	37%	19%	3%	0%
10	0%	0%	0%	1%	6%	16%	23%	23%	17%	9%	5%

Hardly an improvement.

Applying Length Prediction to Next Node Prediction

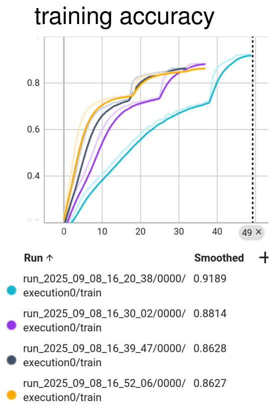
But actually we are not really interested in the path length accuracy per se.

- Real question: is the accuracy sufficient for next node predictions?
  - Consider prediction of next node k in path from node i to node j.
  - If $i = j$ or there is an edge from i to j we are done (no prediction is needed).
  - Otherwise, consider every node connected to node i by an edge.
  - Predict their distances to j and choose some node k with minimum distance.
  - Choice is good, if node k indeed has minimum distance.
    - Even if the actual distance is different from the predicted one.
- Length prediction accuracy (10 nodes, 20 random edges): 0.57.
  - 25,000 samples; 14,353 correct predictions.
- Resulting next node prediction accuracy: in range $[0.79, 0.92]$.
  - 8,367 samples with minimum distance ≥ 2; 7,660 samples with some prediction correct; 6,588 samples with all predictions correct.

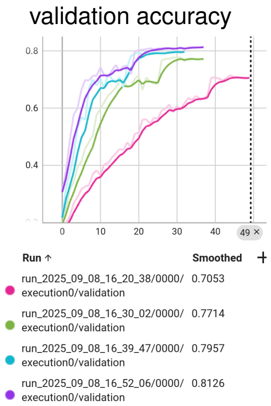
Similar accuracy than with direct next node prediction (but much faster training).

Predicting Next Nodes by Neural Networks

Width $2 + 2 \cdot 10^2$, depth 3, training set sizes $30{,}000$, $60{,}000$, $100{,}000$, $150{,}000$.



Run ↑	Smoothed
● run_2025_09_08_16_20_38/0000/execution0/train	0.9189
● run_2025_09_08_16_30_02/0000/execution0/train	0.8814
● run_2025_09_08_16_39_47/0000/execution0/train	0.8628
● run_2025_09_08_16_52_06/0000/execution0/train	0.8627



Run ↑	Smoothed
● run_2025_09_08_16_20_38/0000/execution0/validation	0.7053
● run_2025_09_08_16_30_02/0000/execution0/validation	0.7714
● run_2025_09_08_16_39_47/0000/execution0/validation	0.7957
● run_2025_09_08_16_52_06/0000/execution0/validation	0.8126

Accuracy is a bit lower than with XGBoost (and training time much longer).

Prediction Accuracy vs Path Lengths

We test the accuracy of predictions for fixed path lengths.

```
print("0:", 1-sum([0.0 if item == 0 else 1.0 for item in model.predict(X_test0).argmax(axis=-1)-y_test0])/len(X_test0))
...
print("10:", 1-sum([0.0 if item == 0 else 1.0 for item in model.predict(X_testx).argmax(axis=-1)-y_testx])/len(X_testx))
```

```
0: 0.99996
1: 0.9337
2: 0.6576
3: 0.73912
4: 0.82498
5: 0.88252
6: 0.9204
7: 0.9532
8: 0.98
9: 0.994
10: 0.45611999999999997
```

Also the neural network model can hardly predict the non-existence of paths.

Input Nodes as Categorical Features

Consider input nodes as “categorical” features rather than as “numerical” ones.

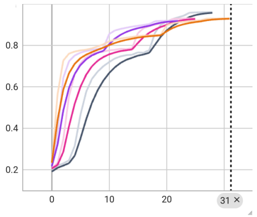
```
def deep_net_cat(width,depth):
    input1 = keras.layers.Input(shape=(1,))
    input2 = keras.layers.Input(shape=(1,))
    input3 = keras.layers.Input(shape=(size*size,))
    encoded1 = keras.layers.CategoryEncoding(num_tokens=size, output_mode="one_hot")(input1)
    encoded2 = keras.layers.CategoryEncoding(num_tokens=size, output_mode="one_hot")(input2)
    inputs = keras.layers.concatenate([encoded1,encoded2,input3])
    layer = inputs
    for _ in range(depth):
      layer = keras.layers.Dense(width, activation="selu", kernel_initializer="lecun_normal")(layer)
    output = keras.layers.Dense(size+1, activation="softmax")(layer)
    return tf.keras.Model(inputs=[input1,input2,input3], outputs=[output])
```

“One-hot encoding” of node i as vector $[0, \ldots, 1, \ldots, 0]$ with single 1 at index i.

Predicting Next Nodes with One-Hot Encoding

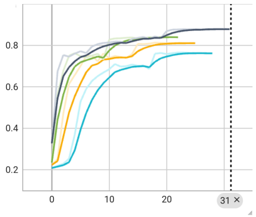
Width $2 \cdot 10 + 2 \cdot 10^2$, depth 3, training set sizes $30{,}000$, $60{,}000$, $100{,}000$, $150{,}000$.

training accuracy



	Run ↑	Smoothed	Value
●	run_2025_09_23_09_55_20/ train	0.9575	0.9609
●	run_2025_09_23_11_52_30/ train	0.9278	0.9308
●	run_2025_09_23_12_02_11/ train	0.9177	0.9202
●	run_2025_09_23_12_13_08/ train	0.929	0.9305

validation accuracy



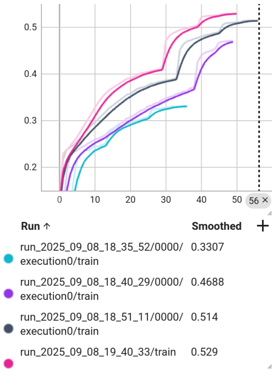
	Run ↑	Smoothed	Value
●	run_2025_09_23_09_55_20/ validation	0.762	0.7616
●	run_2025_09_23_11_52_30/ validation	0.8111	0.8115
●	run_2025_09_23_12_02_11/ validation	0.8396	0.839
●	run_2025_09_23_12_13_08/ validation	0.8785	0.8784

Accuracy is now comparable with that of XGBoost.

Predicting Path Lengths by Neural Network Classification

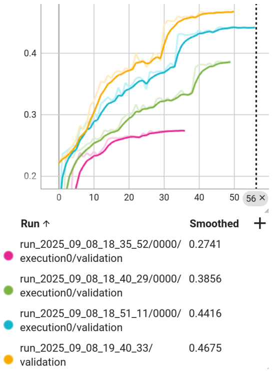
Width $2 + 2 \cdot 10^2$, depth 3, training set sizes $30{,}000$, $60{,}000$, $100{,}000$, $150{,}000$.

training accuracy



	Run ↑	Smoothed
●	run_2025_09_08_18_35_52/0000/ execution0/train	0.3307
●	run_2025_09_08_18_40_29/0000/ execution0/train	0.4688
●	run_2025_09_08_18_51_11/0000/ execution0/train	0.514
●	run_2025_09_08_19_40_33/train	0.529

validation accuracy



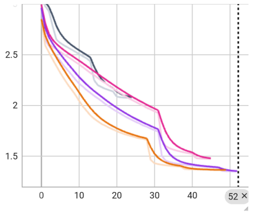
	Run ↑	Smoothed
●	run_2025_09_08_18_35_52/0000/ execution0/validation	0.2741
●	run_2025_09_08_18_40_29/0000/ execution0/validation	0.3856
●	run_2025_09_08_18_51_11/0000/ execution0/validation	0.4416
●	run_2025_09_08_19_40_33/ validation	0.4675

Accuracy is a bit higher than with XGBoost.

Predicting Path Lengths by Neural Network Regression

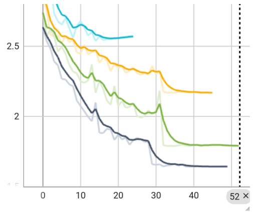
Width $2 + 2 \cdot 10^2$, depth 3, training set sizes $30{,}000$, $60{,}000$, $100{,}000$, $150{,}000$.

training RSME



	Run ↑	Smoothed	Value
●	run_2025_09_09_11_18_45/ train	2.083	2.0727
●	run_2025_09_09_11_22_09/ train	1.4797	1.4732
●	run_2025_09_09_11_32_59/ train	1.3535	1.3495
●	run_2025_09_09_11_47_51/ train	1.3649	1.3644

validation RSME



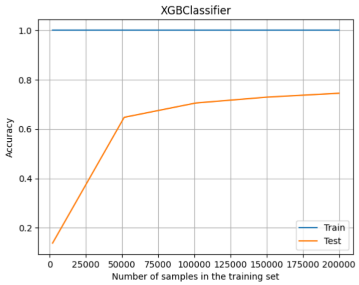
	Run ↑	Smoothed	Value
●	run_2025_09_09_11_18_45/ validation	2.5687	2.5718
●	run_2025_09_09_11_22_09/ validation	2.1698	2.1697
●	run_2025_09_09_11_32_59/ validation	1.7916	1.7915
●	run_2025_09_09_11_47_51/ validation	1.6436	1.6436

Error is a bit lower than with XGBoost.

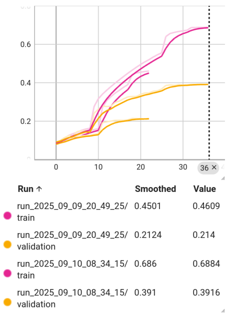
Finally One More (and Larger) Problem

Predicting next nodes in graphs with 20 nodes and 40 random edges.

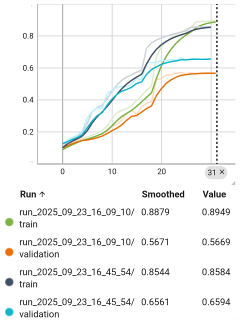
- XGBoost: `XGBClassifier(random_state=42, max_depth=12)`
- Neural network: `deep_net(_cat)(2+`$\underline{1}$`*20*20, 3)`, training set sizes: $250{,}000$, $500{,}000$.



```
CPU times: user 2.5 s, sys: 8.41 s, total: 10.9 s
Wall time: 25min 39s
```



Run ↑	Smoothed	Value
● run_2025_09_09_20_49_25/ train	0.4501	0.4609
● run_2025_09_09_20_49_25/ validation	0.2124	0.214
● run_2025_09_10_08_34_15/ train	0.686	0.6884
● run_2025_09_10_08_34_15/ validation	0.391	0.3916

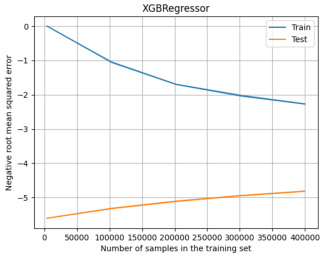


Run ↑	Smoothed	Value
● run_2025_09_23_16_09_10/ train	0.8879	0.8949
● run_2025_09_23_16_09_10/ validation	0.5671	0.5669
● run_2025_09_23_16_45_54/ train	0.8544	0.8584
● run_2025_09_23_16_45_54/ validation	0.6561	0.6594

With `XGBoost` and neural network (with one-hot encoding and much larger training set), still a substantially "higher than chance" accuracy achievable.

Applying Length Prediction to Next Node Prediction

`XGBRegressor(random_state=42, max_depth=20)`



```
CPU times: user 1.55 s, sys: 1.07 s, total: 2.62 s
Wall time: 3min 27s
```

- Length prediction accuracy: 0.47.
  - 25,000 samples; 11,639 predictions correct.
- Resulting next node prediction accuracy: $[0.59, 0.77]$
  - 11,051 samples with minimum distance ≥ 2; 8,524 samples with some prediction correct; 6560 samples with all predictions correct.

Still "higher than chance" accuracy, but lower than with classification.

CONCLUSIONS

Conclusions

So what do I take away from these ML experiments on the shortest path problem?

- Next node prediction with “higher than chance” accuracy seems feasible.
    - Even by training from a minuscle fraction of the problem instance space.
    - Possibly also via path length predictions (lower accuracy but also less training).
- Decision forests (XGBoost) are attractive for this kind of problem.
    - Few hyperparameters, moderate training effort, good accuracy.
- Neural networks (MLPs) are more difficult to utilize.
    - Many hyperparameters, large training effort, mostly not better accuracy.
    - However, accuracy may be slightly superior for path length prediction.

Finally:

- All experiments were based on the supervised learning paradigm.
    - Training on carefully prepared labeled data sets.
- Next stop: reinforcement learning with neural networks.
    - Training “on the fly” while actually performing the path search.