Data Science Documentation

Release 0.1

Jake Teo

Oct 08, 2019

Contents

This documentation summarises various machine learning techniques in Python. A lot of the content are compiled from various resources, so please cite them appropriately if you are using.

CHAPTER 1

General Notes

1.1 Virtual Environment

Every project has a different set of requirements and different set of python packages to support it. The versions of each package can differ or break with each python or dependent packages update, so it is important to isolate every project within an enclosed virtual environment. Anaconda provides a straight forward way to manage this.

```
# create environment
conda create -n yourenvname
# activate environment
source activate yourenvname
# install package
conda install -n yourenvname [package]
# deactivate environment
conda deactivate
# delete environment
conda remove -n yourenvname -all

# see all environments
conda info -e
```

An asterisk (*) will be placed at the current active environment.

```
(chiller)      .gs-Air:~ __,_..g$ conda info -e
# conda environments:
#
base                     /Users/     ıg/anaconda3
chiller               *  /Users/     ıg/anaconda3/envs/chiller
```

Fig. 1: Current active environment

1.2 Modeling

A parsimonious model is a the model that accomplishes the desired level of prediction with as few predictor variables as possible.

1.2.1 Variables

`x` = independent variable = explanatory = predictor

`y` = dependent variable = response = target

1.2.2 Data Types

The type of data is essential as it determines what kind of tests can be applied to it.

`Continuous`: Also known as quantitative. Unlimited number of values

`Categorical`: Also known as discrete or qualitative. Fixed number of values or *categories*

1.2.3 Bias-Variance Tradeoff

The best predictive algorithm is one that has good *Generalization Ability*. With that, it will be able to give accurate predictions to new and previously unseen data.

High Bias results from *Underfitting* the model. This usually results from erroneous assumptions, and cause the model to be too general.

High Variance results from *Overfitting* the model, and it will predict the training dataset very accurately, but not with unseen new datasets. This is because it will fit even the slightless noise in the dataset.

The best model with the highest accuarcy is the middle ground between the two.

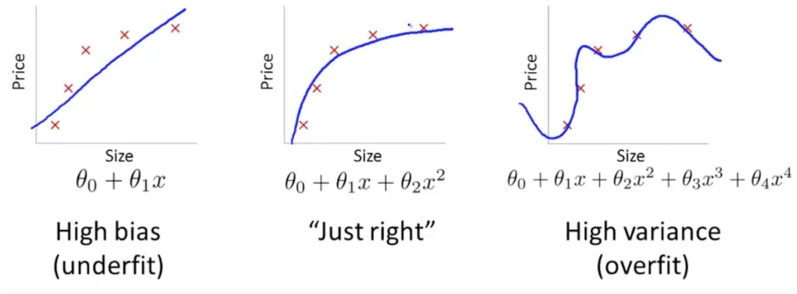


Fig. 2: from Andrew Ng's lecture

1.2.4 Steps to Build a Predictive Model

Feature Selection, Preprocessing, Extraction

1. Remove features that have too many NAN or fill NAN with another value
2. Remove features that will introduce data leakage

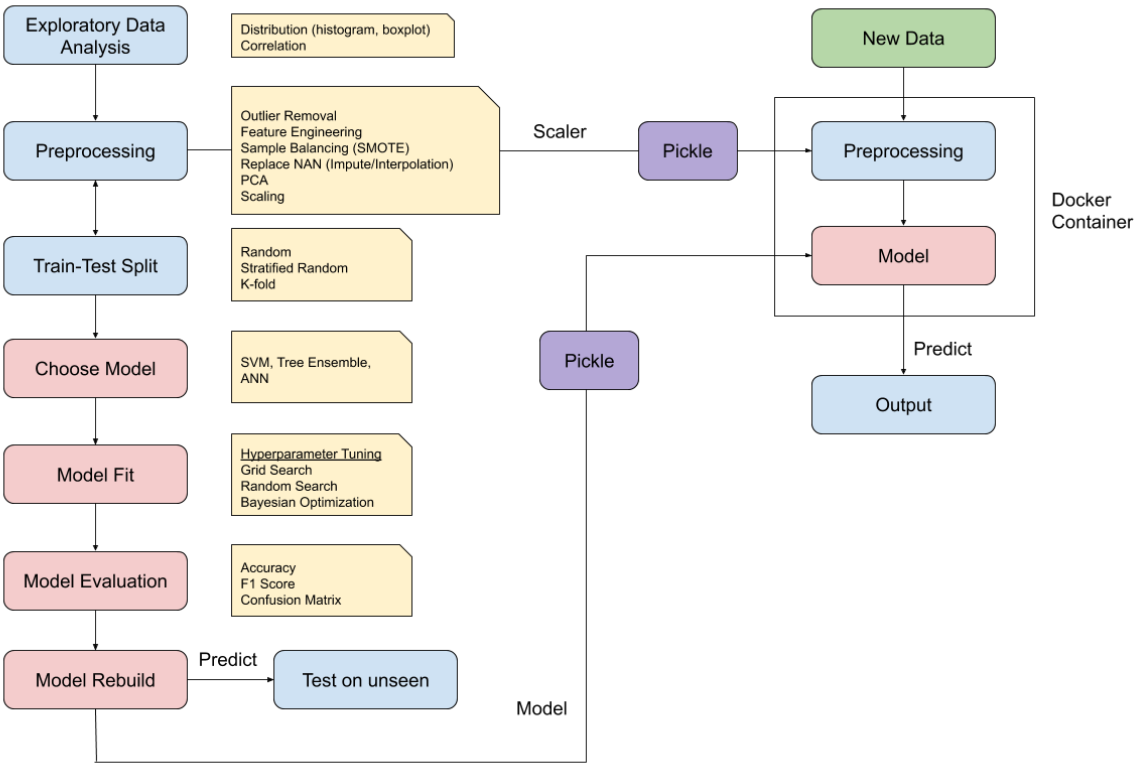


Fig. 3: Typical architecture for model building for supervised classification

3. Encode categorical features into integers
4. Extract new useful features (between and within current features)

Normalise the Features

With the exception of Tree models and Naive Bayes, other machine learning techniques like Neural Networks, KNN, SVM should have their features scaled.

Train Test Split

Split the dataset into *Train* and *Test* datasets. By default, sklearn assigns 75% to train & 25% to test randomly. A random state (seed) can be selected to fixed the randomisation

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test
= train_test_split(predictor, target, test_size=0.25, random_state=0)
```

Create Model

Choose model and set model parameters (if any).

```
clf = DecisionTreeClassifier()
```

Fit Model

Fit the model using the training dataset.

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

```
>>> print model
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')
```

Test Model

Test the model by predicting identity of unseen data using the testing dataset.

```
y_predict = model.predict(X_test)
```

Score Model

Use a confusion matrix and...

```
>>> print sklearn.metrics.confusion_matrix(y_test, predictions)
[[14  0  0]
 [ 0 13  0]
 [ 0  1 10]]
```

accuarcy percentage, and f1 score to obtain the predictive accuarcy.

```
import sklearn.metrics
print sklearn.metrics.accuracy_score(y_test, y_predict)*100, '%'
>>> 97.3684210526 %
```

Cross Validation

When all code is working fine, remove the train-test portion and use Grid Search Cross Validation to compute the best parameters with cross validation.

Final Model

Finally, rebuild the model using the full dataset, and the chosen parameters tested.

1.2.5 Quick-Analysis for Multi-Models

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from xgboost import XGBClassifier

from sklearn.metrics import accuracy_score, f1_score
from statistics import mean
import seaborn as sns

# models to test
svml = LinearSVC()
svm = SVC()
rf = RandomForestClassifier()
xg = XGBClassifier()
xr = ExtraTreesClassifier()

# iterations
classifiers = [svml, svm, rf, xr, xg]
names = ['Linear SVM', 'RBF SVM', 'Random Forest', 'Extremely Randomized Trees',
↪'XGBoost']
results = []

# train-test split
X = df[df.columns[:-1]]
# normalise data for SVM
X = StandardScaler().fit(X).transform(X)
```

(continues on next page)

(continued from previous page)

```
y = df['label']
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

for name, clf in zip(names, classifiers):
    model = clf.fit(X_train, y_train)
    y_predict = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_predict)
    f1 = mean(f1_score(y_test, y_predict, average=None))
    results.append([fault, name, accuracy, f1])
```

A final heatmap to compare the outcomes.

```
final = pd.DataFrame(results, columns=['Fault Type','Model','Accuracy','F1 Score'])
final.style.background_gradient(cmap='Greens')
```

	Model	Accuracy	F1 Score
0	Linear SVM	0.84	0.832134
1	RBF SVM	0.687692	0.66756
2	Random Forest	0.912308	0.910632
3	Extremely Randomized Trees	0.870769	0.868625
4	XGBoost	0.938462	0.93719

CHAPTER 2

In-Built Datasets

There are in-built datasets provided in both statsmodels and sklearn packages.

Statsmodels

In statsmodels, many R datasets can be obtained from the function `sm.datasets.get_rdataset()`. To view each dataset's description, use `print(duncan_prestige.__doc__)`.

```
import statsmodels.api as sm
prestige = sm.datasets.get_rdataset("Duncan", "car", cache=True).data
print prestige.head()
```

```
type  income  education  prestige
accountant  prof      62         86        82
pilot       prof      72         76        83
architect   prof      75         92        90
author      prof      55         90        76
chemist     prof      64         86        90
```

Sklearn

There are five common toy datasets here. For others, view http://scikit-learn.org/stable/datasets/index.html. To view each dataset's description, use `print boston['DESCR']`.

load_boston([return_X_y])	Load and return the boston house-prices dataset (regression).
load_iris([return_X_y])	Load and return the iris dataset (classification).
load_diabetes([return_X_y])	Load and return the diabetes dataset (regression).
load_digits([n_class, return_X_y])	Load and return the digits dataset (classification).
load_linnerud([return_X_y])	Load and return the linnerud dataset (multivariate regression).

```
from sklearn.datasets import load_iris

# Load Iris data (https://en.wikipedia.org/wiki/Iris_flower_data_set)
iris = load_iris()
# Load iris into a dataframe and set the field names
df = pd.DataFrame(iris['data'], columns=iris['feature_names'])
df.head()

sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2



# Feature names are in .target & .target_names
>>> print iris.target_names[:5]
>>> ['setosa' 'versicolor' 'virginica']
>>> print iris.target
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]



# Change target to target_names & merge with main dataframe
df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)
print df['species'].head()

sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
0    setosa
1    setosa
2    setosa
3    setosa
4    setosa
Name: species, dtype: category
Categories (3, object): [setosa, versicolor, virginica]
```

CHAPTER 3

Exploratory Analysis

Exploratory data analysis (EDA) is an essential step to understand the data better; in order to engineer and select features before modelling. This often requires skills in visualisation to better interpret the data.

3.1 Univariate

3.1.1 Distribution Plots

When plotting distributions, it is important to compare the distribution of both train and test sets. If the test set very specific to certain features, the model will underfit and have a low accuarcy.

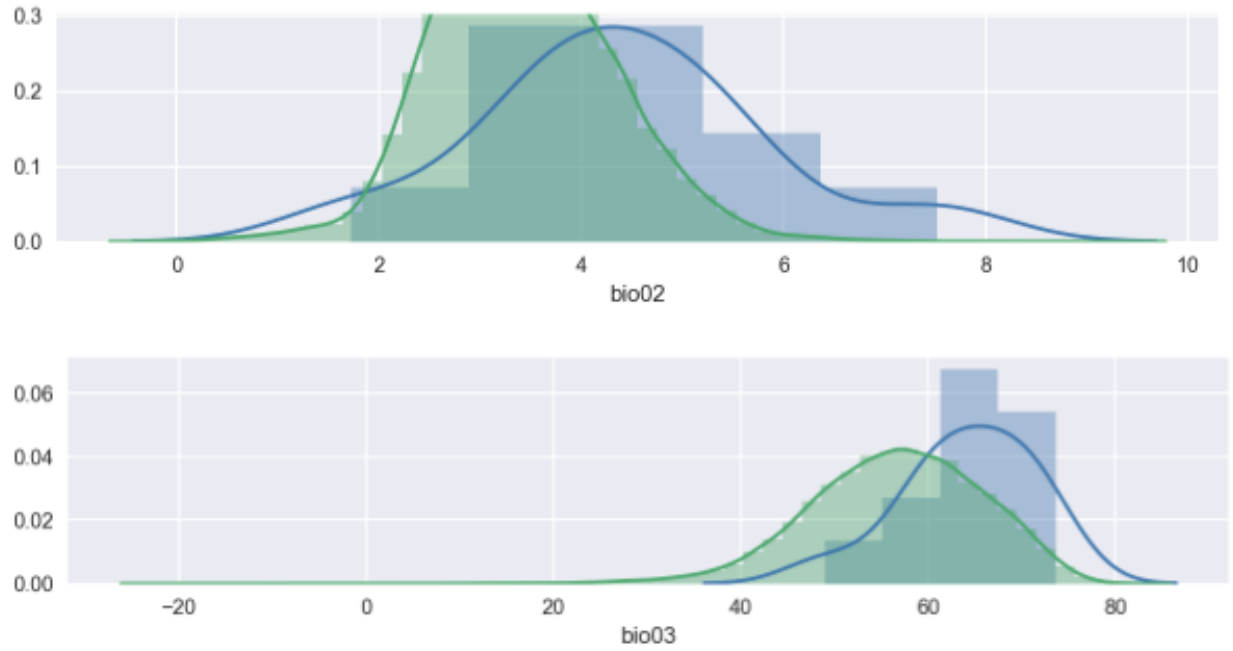
```
import seaborn as sns
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
%matplotlib inline

for i in X.columns:
    plt.figure(figsize=(15,5))
    sns.distplot(X[i])
    sns.distplot(pred[i])
```

3.1.2 Count Plots

For **categorical** features, you may want to see if they have enough sample size for each category.

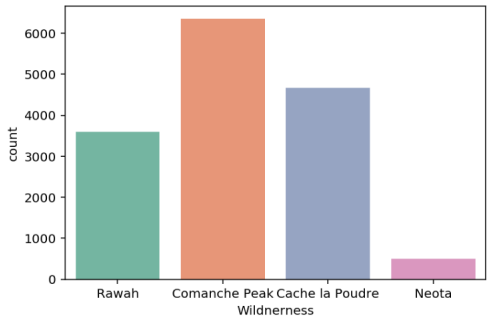
```
import seaborn as sns
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
%matplotlib inline

df['Wildnerness'].value_counts()
```

(continues on next page)



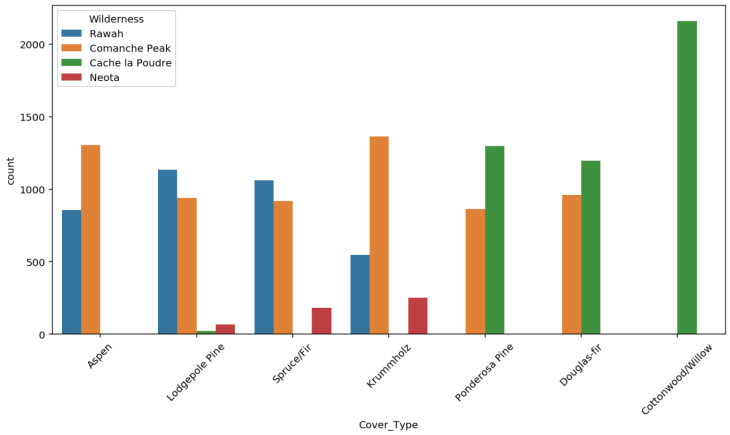
(continued from previous page)

```
Comanche Peak      6349
Cache la Poudre    4675
Rawah              3597
Neota               499
Name: Wildnerness, dtype: int64

cmap = sns.color_palette("Set2")
sns.countplot(x='Wildnerness',data=df, palette=cmap);
plt.xticks(rotation=45);
```



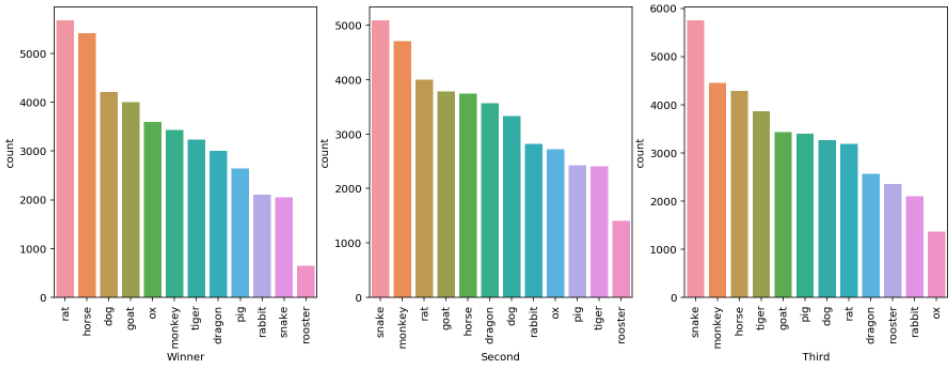
To check for possible relationships with the target, place the feature under hue.

```
plt.figure(figsize=(12,6))
sns.countplot(x='Cover_Type',data=wild, hue='Wilderness');
plt.xticks(rotation=45);
```



Multiple Plots

```
fig, axes = plt.subplots(ncols=3, nrows=1, figsize=(15, 5)) # note only for 1 row or␣
↪1 col, else need to flatten nested list in axes
col = ['Winner','Second','Third']

for cnt, ax in enumerate(axes):
    sns.countplot(x=col[cnt], data=df2, ax=ax, order=df2[col[cnt]].value_counts().
↪index);

for ax in fig.axes:
    plt.sca(ax)
    plt.xticks(rotation=90)
```

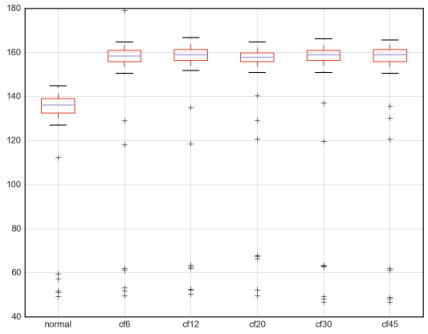


3.1.3 Box Plots

Using the 50 percentile to compare among different classes, it is easy to find feature that can have high prediction importance if they do not overlap. Also can be use for outlier detection. Features have to be **continuous**.

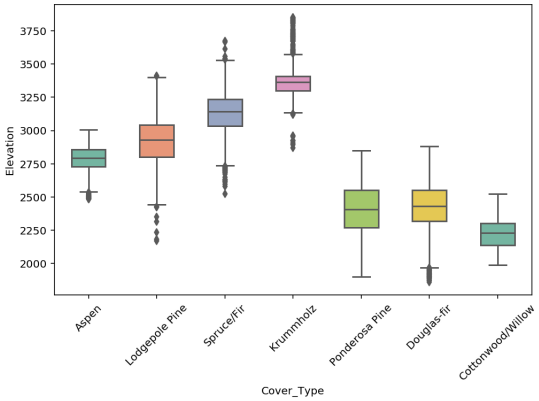
From different dataframes, displaying the same feature.

```
df = pd.DataFrame({'normal': normal['Pressure'], 's1': cf6['Pressure'], 's2': cf12[
↪'Pressure'],
                   's3': cf20['Pressure'], 's4': cf30['Pressure'],'s5': cf45[
↪'Pressure']})
df.boxplot(figsize=(10,5));
```



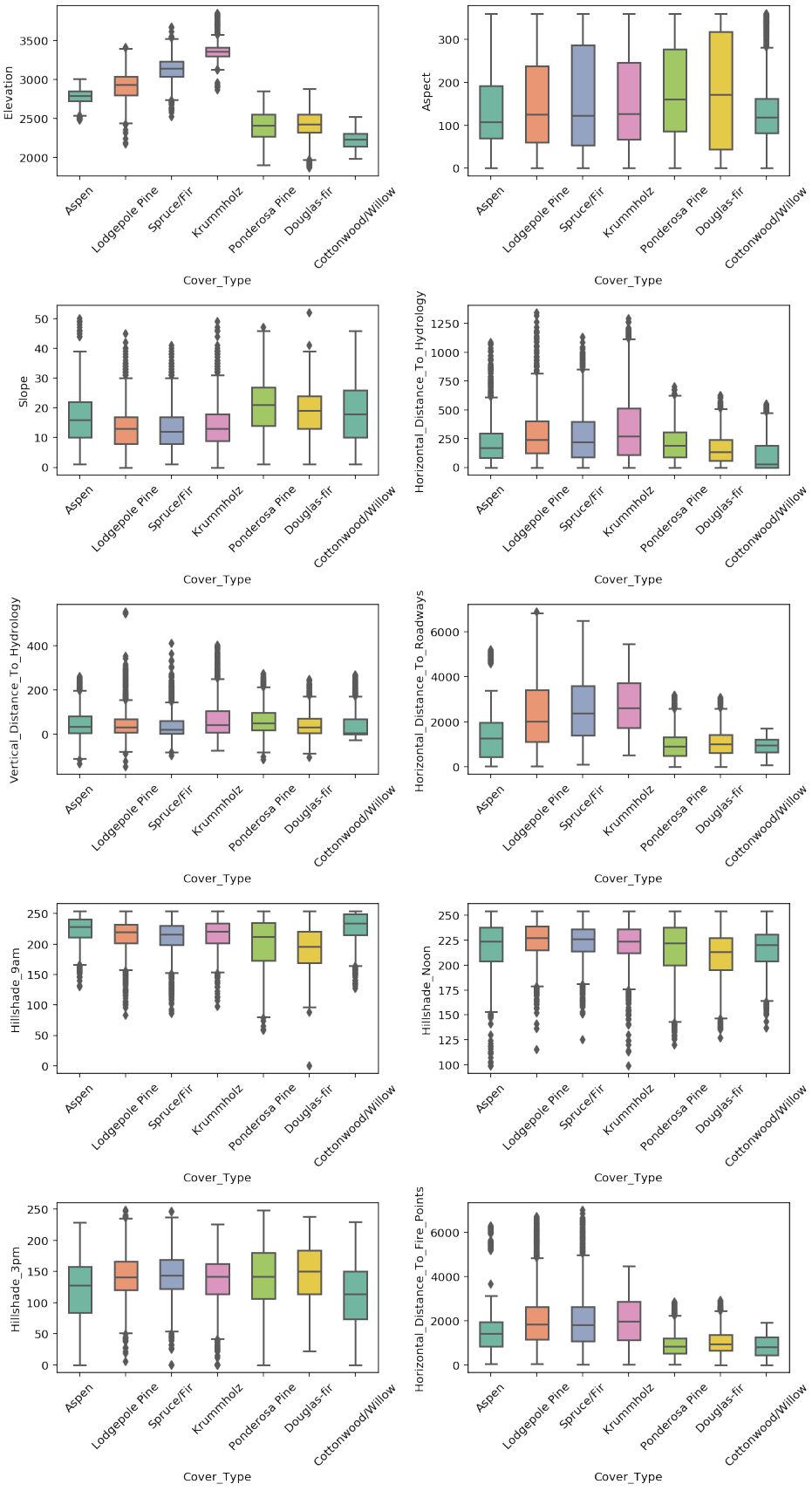
From same dataframe with of a feature split by different y-labels

```
plt.figure(figsize=(7, 5))
cmap = sns.color_palette("Set3")
sns.boxplot(x='Cover_Type', y='Elevation', data=df, palette=cmap);
plt.xticks(rotation=45);
```



Multiple Plots

```
cmap = sns.color_palette("Set2")

fig, axes = plt.subplots(ncols=2, nrows=5, figsize=(10, 18))
a = [i for i in axes for i in i] # axes is nested if >1 row & >1 col, need to flatten
for i, ax in enumerate(a):
    sns.boxplot(x='Cover_Type', y=eda2.columns[i], data=eda, palette=cmap, width=0.5,␣
↪ax=ax);

# rotate x-axis for every single plot
for ax in fig.axes:
    plt.sca(ax)
    plt.xticks(rotation=45)

# set spacing for every subplot, else x-axis will be covered
plt.tight_layout()
```


Elevation
Aspect
Slope
Horizontal_Distance_To_Hydrology
Vertical_Distance_To_Hydrology
Horizontal_Distance_To_Roadways
Hillshade_9am
Hillshade_Noon
Hillshade_3pm
Horizontal_Distance_To_Fire_Points
Aspen
Lodgepole Pine
Spruce/Fir
Krummholz
Ponderosa Pine
Douglas-fir
Cottonwood/Willow
Cover_Type

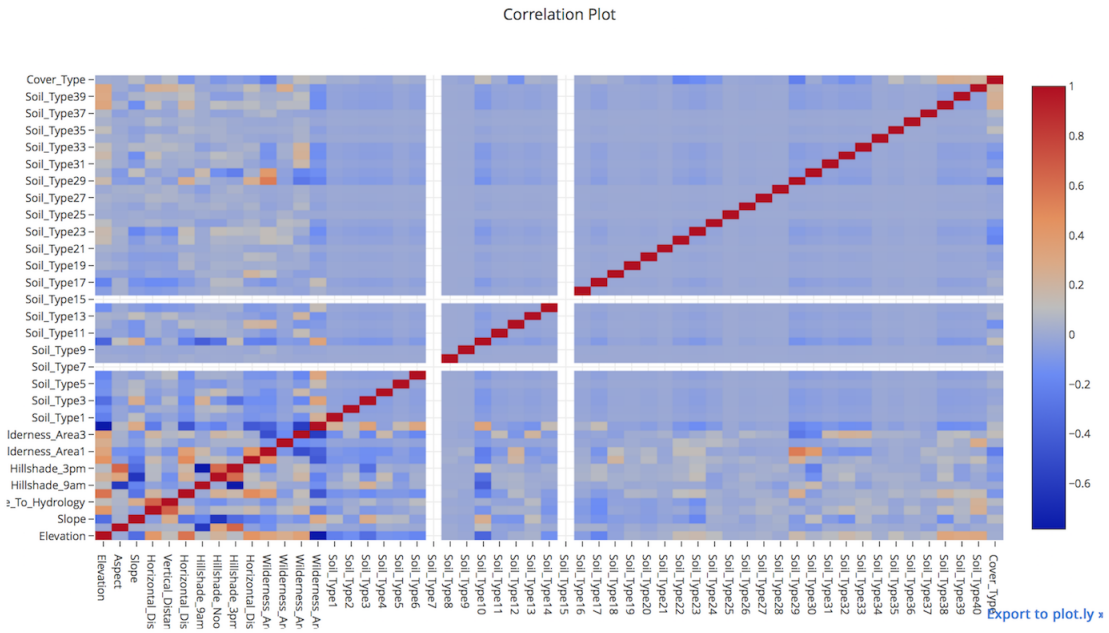
3.2 Multi-Variate

3.2.1 Correlation Plots

Heatmaps show a quick overall correlation between features.

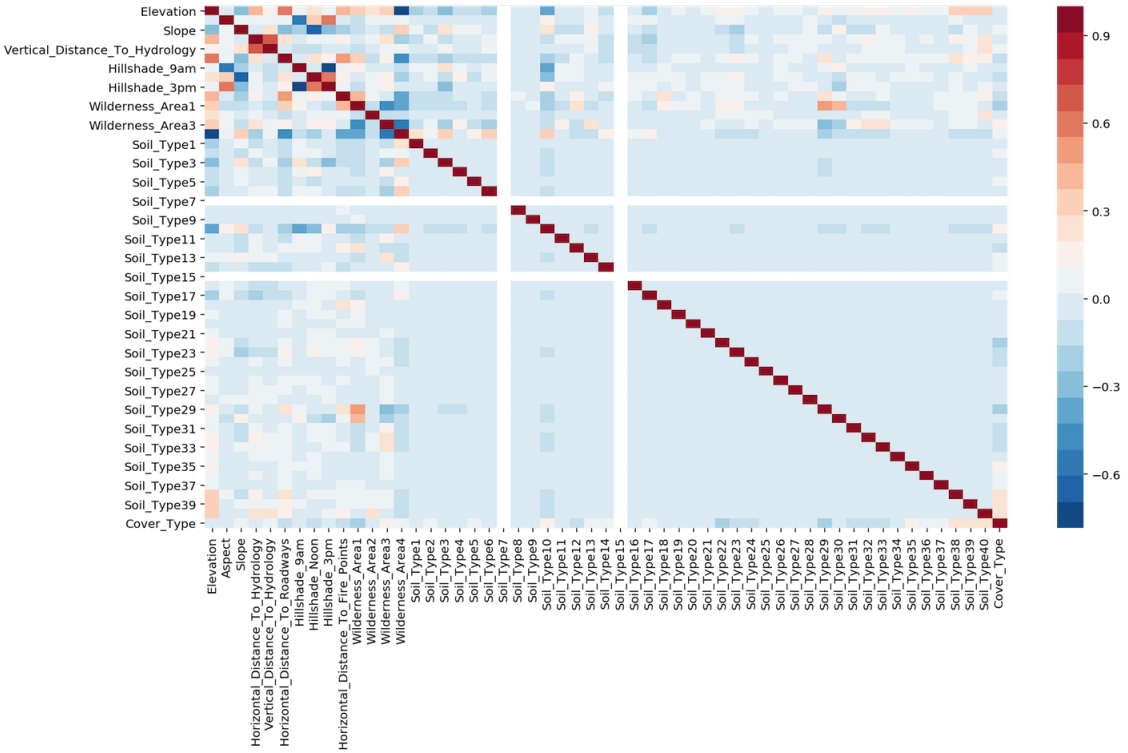
Using plot.ly

```
from plotly.offline import iplot
from plotly.offline import init_notebook_mode
import plotly.graph_objs as go
init_notebook_mode(connected=True)

# create correlation in dataframe
corr = df[df.columns[1:]].corr()

layout = go.Layout(width=1000, height=600, \
                   title='Correlation Plot', \
                   font=dict(size=10))
data = go.Heatmap(z=corr.values, x=corr.columns, y=corr.columns)
fig = go.Figure(data=[data], layout=layout)
iplot(fig)
```



Using seaborn

```
import seaborn as sns
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
%matplotlib inline

# create correlation in dataframe
corr = df[df.columns[1:]].corr()

plt.figure(figsize=(15, 8))
sns.heatmap(corr, cmap=sns.color_palette("RdBu_r", 20));
```


Elevation
Slope
Vertical_Distance_To_Hydrology
Hillshade_9am
Hillshade_3pm
Wilderness_Area1
Wilderness_Area3
Soil_Type1
Soil_Type3
Soil_Type5
Soil_Type7
Soil_Type9
Soil_Type11
Soil_Type13
Soil_Type15
Soil_Type17
Soil_Type19
Soil_Type21
Soil_Type23
Soil_Type25
Soil_Type27
Soil_Type29
Soil_Type31
Soil_Type33
Soil_Type35
Soil_Type37
Soil_Type39
Cover_Type
Elevation
Aspect
Slope
Horizontal_Distance_To_Hydrology
Vertical_Distance_To_Hydrology
Horizontal_Distance_To_Roadways
Hillshade_9am
Hillshade_Noon
Hillshade_3pm
Horizontal_Distance_To_Fire_Points
Wilderness_Area1
Wilderness_Area2
Wilderness_Area3
Wilderness_Area4
Soil_Type1
Soil_Type2
Soil_Type3
Soil_Type4
Soil_Type5
Soil_Type6
Soil_Type7
Soil_Type8
Soil_Type9
Soil_Type10
Soil_Type11
Soil_Type12
Soil_Type13
Soil_Type14
Soil_Type15
Soil_Type16
Soil_Type17
Soil_Type18
Soil_Type19
Soil_Type20
Soil_Type21
Soil_Type22
Soil_Type23
Soil_Type24
Soil_Type25
Soil_Type26
Soil_Type27
Soil_Type28
Soil_Type29
Soil_Type30
Soil_Type31
Soil_Type32
Soil_Type33
Soil_Type34
Soil_Type35
Soil_Type36
Soil_Type37
Soil_Type38
Soil_Type39
Soil_Type40
Cover_Type
0.9
0.6
0.3
0.0
−0.3
−0.6

CHAPTER 4

Feature Preprocessing

4.1 Numeric

4.1.1 Feature Proprocessing

Missing Values

We can change missing values for the entire dataframe into their individual column means or medians.

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer

impute = SimpleImputer(missing_values=np.nan, strategy='median', copy=False)
imp_mean.fit(df)
# output is in numpy, so convert to df
df2 = pd.DataFrame(imp_mean.transform(df),columns=df.columns)
```

We can also use interpolation via pandas default function to fill in the missing values. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.interpolate.html

```
import pandas as pd

df['colname'].interpolate(method='linear', limit=2)
```

Scaling

Many ML algorithms require normalization or scaling. See more in here.

Outliers

Especially sensitive in linear models. They can be (1) removed manually by defining the lower and upper bound limit, or (2) grouping the features into ranks.

Transformation

This helps in non-tree based and especially neural networks. Helps to drive big values close to features' average value.

Using Log Transform `np.log(1+x)`. Or Raising to the power of one `np.sqrt(x+2/3)`

Another important moment which holds true for all preprocessings is that sometimes, it is beneficial to train a model on concatenated data frames produced by different preprocessings, or to mix models training differently-preprocessed data. Again, linear models, KNN, and neural networks can benefit hugely from this.

4.1.2 Feature Generation

Sometimes, we can engineer these new features using *prior knowledge and logic*, or *using Exploratory Data Analysis*.

Examples include:

- multiplication, divisions, addition, and feature interactions
- feature extraction

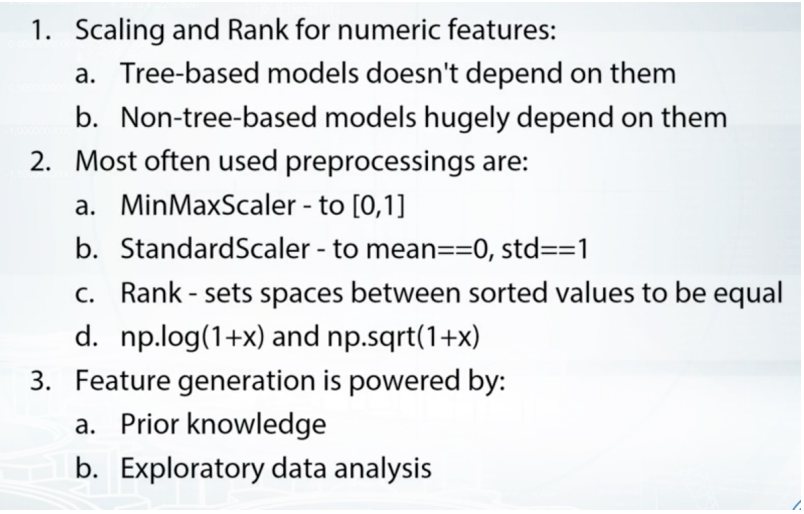


Fig. 1: Coursera: How to Win a Data Science Competition

4.2 Categorical & Ordinal

Ordinal features are categorical but ranked in a meaningful way.

4.2.1 Tree-Based Models

Label Encoding: or conversion of category into integers.

- Alphabetical order `sklearn.preprocessing.LabelEncoder`

- Order of appearance `pd.factorize`

```
from sklearn import preprocessing

# Test data
df = DataFrame(['A', 'B', 'B', 'C'], columns=['Col'])

df['Fact'] = pd.factorize(df['Col'])[0]

le = preprocessing.LabelEncoder()
df['Lab'] = le.fit_transform(df['Col'])

print(df)
#   Col  Fact  Lab
# 0   A     0    0
# 1   B     1    1
# 2   B     1    1
# 3   C     2    2
```

Frequency Encoding: conversion of category into frequencies.

```
### FREQUENCY ENCODING

# size of each category
encoding = titanic.groupby('Embarked').size()
# get frequency of each category
encoding = encoding/len(titanic)
titanic['enc'] = titanic.Embarked.map(encoding)

# if categories have same frequency it can be an issue
# will need to change it to ranked frequency encoding
from scipy.stats import rankdata
```

4.2.2 Non-Tree Based Models

One-Hot Encoding: We could use an integer encoding directly, rescaled where needed. This may work for problems where there is a natural ordinal relationship between the categories, and in turn the integer values, such as labels for temperature 'cold', warm', and 'hot'. There may be problems when there is no *ordinal* relationship and allowing the representation to lean on any such relationship might be damaging to learning to solve the problem. An example might be the labels 'dog' and 'cat'.

Each category is one binary field of 1 & 0. Not good if too many categories in a feature. Need to store in sparse matrix.

- Dummies: `pd.get_dummies`, this converts a string into binary, and splits the columns according to n categories
- sklearn: `sklearn.preprocessing.OneHotEncoder`, string has to be converted into numeric, then stored in a sparse matrix.

Feature Interactions: interactions btw categorical features

- Linear Models & KNN

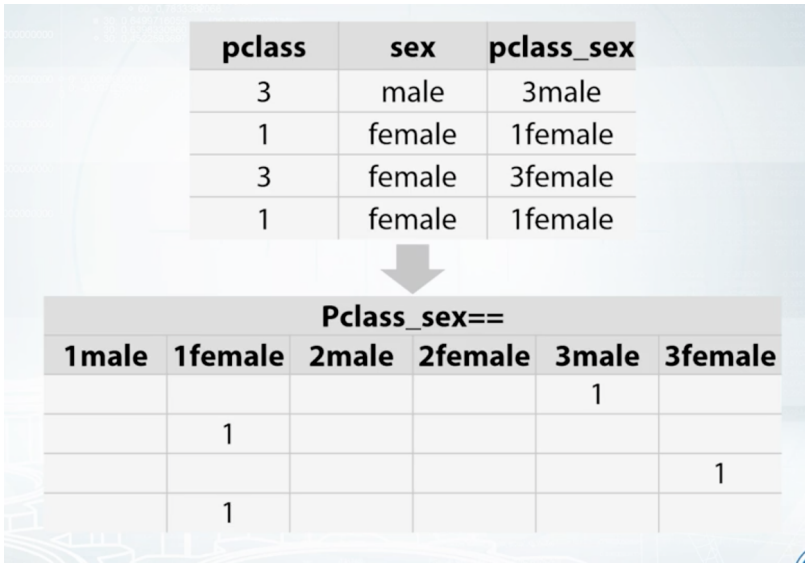


Fig. 2: Coursera: How to Win a Data Science Competition

4.3 Datetime

4.4 Coordinates

It is necessary to define a projection for a coordinate reference system if there is a classification in space, eg k-means clustering. This basically change the coordinates from a spherical component to a flat surface.

Also take note of spatial auto-correlation.

CHAPTER 5

Feature Normalization

Normalisation is another important concept needed to change all features to the same scale. This allows for faster convergence on learning, and more uniform influence for all weights. More on sklearn website:

- http://scikit-learn.org/stable/modules/preprocessing.html

Tree-based models is not dependent on scaling, but non-tree models models, very often are hugely dependent on it.

Outliers can affect certain scalers, and it is important to either remove them or choose a scalar that is robust towards them.

- https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html
- http://benalexkeen.com/feature-scaling-with-scikit-learn/

5.1 Scaling

5.1.1 Standard Scaler

It standardize features by removing the mean and scaling to unit variance The standard score of a sample x is calculated as:

z = (x - u) / s

```
import pandas pd
from sklearn.preprocessing import StandardScaler

X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                    random_state = 0)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
# note that the test set using the fitted scaler in train dataset to transform in the␣
↪test set
X_test_scaled = scaler.transform(X_test)
```

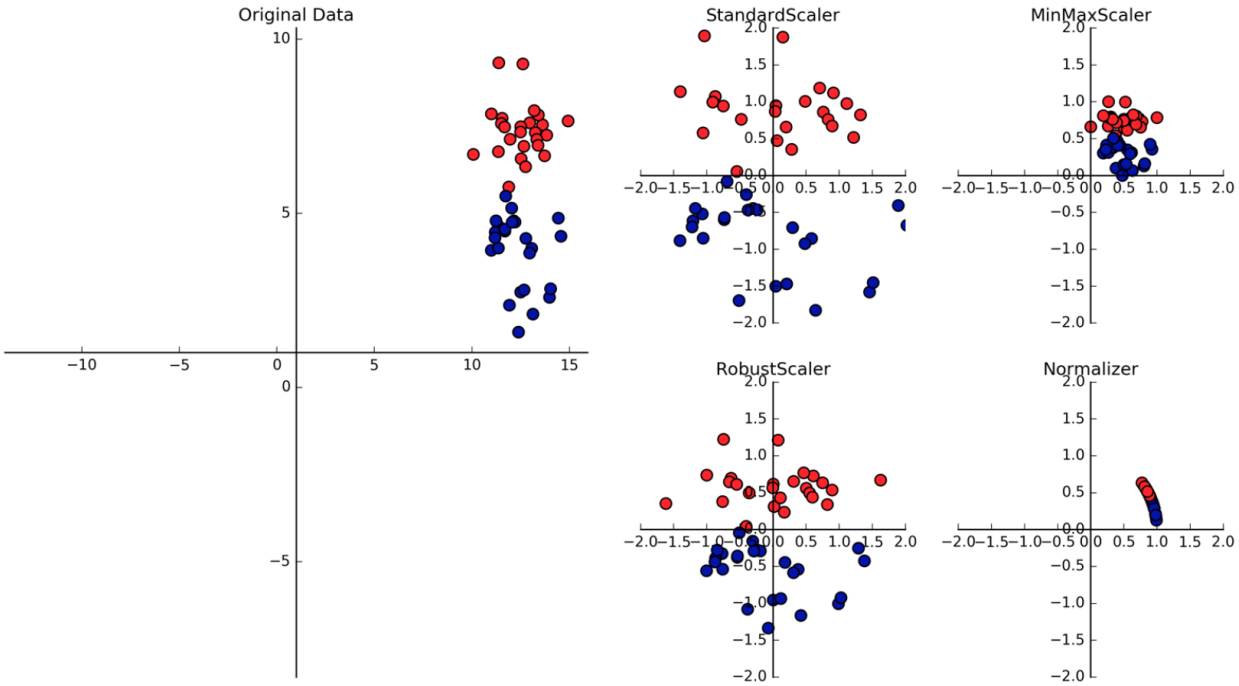


Fig. 1: Introduction to Machine Learning in Python

5.1.2 Min Max Scale

Another way to normalise is to use the Min Max Scaler, which changes all features to be between 0 and 1, as defined below:

$$x_i' = (x_i - x_i^{MIN})/(x_i^{MAX} - x_i^{MIN})$$

```
import pandas pd
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                    random_state = 0)

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

linridge = Ridge(alpha=20.0).fit(X_train_scaled, y_train)
```

5.1.3 RobustScaler

Works similarly to standard scaler except that it uses median and quartiles, instead of mean and variance. Good as it ignores data points that are outliers.

5.1.4 Normalizer

Scales each data point such that the feature vector has a Euclidean length of 1. Often used when the direction of the data matters, not the length of the feature vector.

5.2 Pipeline

Scaling have a chance of leaking the part of the test data in train-test split into the training data. This is especially inevitable when using cross-validation.

We can scale the train and test datasets separately to avoid this. However, a more convenient way is to use the pipeline function in sklearn, which wraps the scaler and classifier together, and scale them separately during cross validation.

Any other functions can also be input here, e.g., rolling window feature extraction, which also have the potential to have data leakage.

```
from sklearn.pipeline import Pipeline

# "scaler" & "svm" can be any name. But they must be placed in the correct order of␣
↪processing
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])

pipe.fit(X_train, y_train)
Pipeline(steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svm',␣
↪SVC(C=1.0, cac
          decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False))])

pipe.score(X_test, y_test)
0.95104895104895104
```

5.3 Persistance

To save the fitted scaler to normalize new datasets, we can save it using pickle or joblib for reusing in the future.

CHAPTER 6

Feature Engineering

Feature Engineering is one of the most important part of model building. Collecting and creating of relevant features from existing ones are most often the determinant of a high prediction value.

They can be classified broadly as:

- **Aggregations**
  - Rolling/sliding Window (overlapping)
  - Tumbling Window (non-overlapping)
- Transformations
- Decompositions
- Interactions

6.1 Manual

6.1.1 Decomposition

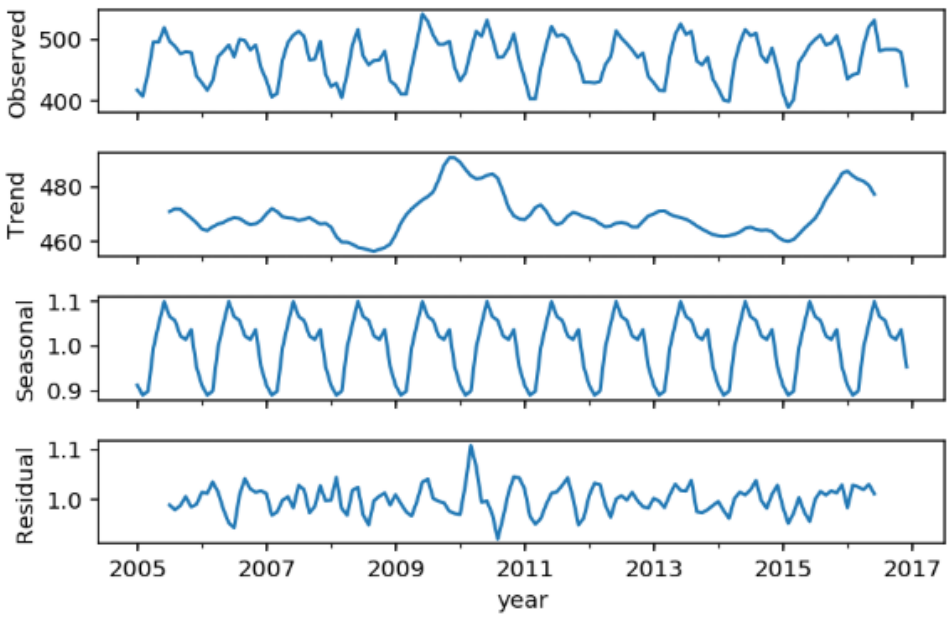
Time-Series

Decomposing a time-series into trend (long-term), seaonality (short-term), residuals (noise). There are two methods to decompose:

- Additive—The component is present and is added to the other components to create the overall forecast value.
- Multiplicative—The component is present and is multiplied by the other components to create the overall forecast value

Usually an additive time-series will be used if there are no seasonal variations over time.

```
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

res = sm.tsa.seasonal_decompose(final2['avg_mth_elect'], model='multiplicative')
res.plot();
```




```
# set decomposed parts into dataframe
decomp=pd.concat([res.observed, res.trend, res.seasonal, res.resid], axis=1)
decomp.columns = ['avg_mth','trend','seasonal','residual']
decomp.head()
```

Fourier Transformation

The Fourier transform (FT) decomposes a function of time (a signal) into its constituent frequencies, i.e., converts amplitudes into frequencies.

Wavelet Transform

Wavelet transforms are time-frequency transforms employing wavelets. They are similar to Fourier transforms, the difference being that Fourier transforms are localized only in frequency instead of in time and frequency. There are various considerations for wavelet transform, including:

- Which wavelet transform will you use, CWT or DWT?
- Which wavelet family will you use?
- Up to which level of decomposition will you go?
- Number of coefficients (vanishing moments)

- What is the right range of scales to use?
- http://ataspinar.com/2018/12/21/a-guide-for-using-the-wavelet-transform-in-machine-learning/
- https://www.kaggle.com/jackvial/dwt-signal-denoising
- https://www.kaggle.com/tarunpaparaju/lanl-earthquake-prediction-signal-denoising

```
import pywt

# there are 14 wavelets families
print(pywt.families(short=False))
#['Haar', 'Daubechies', 'Symlets', 'Coiflets', 'Biorthogonal', 'Reverse biorthogonal',
#'Discrete Meyer (FIR Approximation)', 'Gaussian', 'Mexican hat wavelet', 'Morlet␣
↪wavelet',
#'Complex Gaussian wavelets', 'Shannon wavelets', 'Frequency B-Spline wavelets',
↪'Complex Morlet wavelets']

# short form used in pywt
print(pywt.families())
#['haar', 'db', 'sym', 'coif', 'bior', 'rbio',
#'dmey', 'gaus', 'mexh', 'morl',
#'cgau', 'shan', 'fbsp', 'cmor']

# input wavelet family, coefficient no., level of decompositions
arrays = pywt.wavedec(array, 'sym5', level=5)
df3 = pd.DataFrame(arrays).T

# gives two arrays, decomposed & residuals
decompose, residual = pywt.dwt(signal,'sym5')
```

6.2 Auto

Automatic generation of new features from existing ones are starting to gain popularity, as it can save a lot of time.

6.2.1 Tsfresh

tsfresh is a feature extraction package for time-series. It can extract more than 1200 different features, and filter out features that are deemed relevant. In essence, it is a univariate feature extractor.

https://tsfresh.readthedocs.io/en/latest/

Extract all possible features

```
from tsfresh import extract_features

def list_union_df(fault_list):
'''
Description
------------
Convert list of faults with a single signal value into a dataframe with an id for␣
↪each fault sample
Data transformation prior to feature extraction
'''
# convert nested list into dataframe
dflist = []
```

(continues on next page)

(continued from previous page)

```
# give an id field for each fault sample
for a, i in enumerate(verified_faults):
    df = pd.DataFrame(i)
    df['id'] = a
    dflist.append(df)

df = pd.concat(dflist)
return df

df = list_union_df(fault_list)

# tsfresh
extracted_features = extract_features(df, column_id='id')
# delete columns which only have one value for all rows
for i in extracted_features.columns:
    col = extracted_features[i]
    if len(col.unique()) == 1:
        del extracted_features[i]
```

Generate only relevant features

```
from tsfresh import extract_relevant_features

# y = is the target vector
    # length of y = no. of samples in timeseries, not length of the entire timeseries
# column_sort = for each sample in timeseries, time_steps column will restart
# fdr_level = false discovery rate, is default at 0.05,
    # it is the expected percentage of irrelevant features
    # tune down to reduce number of created features retained, tune up to increase

features_filtered_direct = extract_relevant_features(timeseries, y,
                                                     column_id='id',
                                                     column_sort='time_steps',
                                                     fdr_level=0.05)
```

6.2.2 FeatureTools

FeatureTools is extremely useful if you have datasets with a base data, with other tables that have relationships to it.

We first create an **EntitySet**, which is like a database. Then we create **entities**, i.e., individual tables with a unique id for each table, and showing their **relationships** between each other.

https://github.com/Featuretools/featuretools

```
import featuretools as ft

def make_entityset(data):
es = ft.EntitySet('Dataset')
es.entity_from_dataframe(dataframe=data,
                         entity_id='recordings',
                         index='index',
                         time_index='time')

es.normalize_entity(base_entity_id='recordings',
                    new_entity_id='engines',
```

(continues on next page)

(continued from previous page)

```
                    index='engine_no')

es.normalize_entity(base_entity_id='recordings',
                    new_entity_id='cycles',
                    index='time_in_cycles')
return es
es = make_entityset(data)
es
```

We then use something called **Deep Feature Synthesis (dfs)** to generate features automatically.

Primitives are the type of new features to be extracted from the datasets. They can be **aggregations** (data is combined) or **transformation** (data is changed via a function) type of extractors. The list can be found via `ft.primitives.list_primitives()`. External primitives like tsfresh, or custom calculations can also be input into FeatureTools.

```
feature_matrix, feature_names = ft.dfs(entityset=es,
                                       target_entity = 'normal',
                                       agg_primitives=['last', 'max', 'min'],
                                       trans_primitives=[],
                                       max_depth = 2,
                                       verbose = 1,
                                       n_jobs = 3)
# see all old & new features created
feature_matrix.columns
```

FeatureTools appears to be a very powerful auto-feature extractor. Some resources to read further are as follows:

- https://brendanhasz.github.io/2018/11/11/featuretools
- https://towardsdatascience.com/automated-feature-engineering-in-python-99baf11cc219
- https://medium.com/@rrfd/simple-automatic-feature-engineering-using-featuretools-in-python-for-classification-b1308040e183

CHAPTER 7

Class Imbalance

In domains like predictive maintenance, machine failures are usually rare occurrences in the lifetime of the assets compared to normal operation. This causes an imbalance in the label distribution which usually causes poor performance as algorithms tend to classify majority class examples better at the expense of minority class examples as the total misclassification error is much improved when majority class is labeled correctly. Techniques are available to correct for this.

The imbalance-learn package provides an excellent range of algorithms for adjusting for imbalanced data. Install with `pip install -U imbalanced-learn` or `conda install -c conda-forge imbalanced-learn`.

An important thing to note is that **resampling must be done AFTER the train-test split**, so as to prevent data leakage.

7.1 Over-Sampling

SMOTE (synthetic minority over-sampling technique) is a common and popular up-sampling technique.

```
from imblearn.over_sampling import SMOTE

smote = SMOTE()
X_resampled, y_resampled = smote.fit_sample(X_train, y_train)
clf = LogisticRegression()
clf.fit(X_resampled, y_resampled)
```

ADASYN is one of the more advanced over sampling algorithms.

```
from imblearn.over_sampling import ADASYN

ada = ADASYN()
X_resampled, y_resampled = ada.fit_sample(X_train, y_train)
clf = LogisticRegression()
clf.fit(X_resampled, y_resampled)
```

7.2 Under-Sampling

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler()
X_resampled, y_resampled = rus.fit_sample(X_train, y_train)
clf = LogisticRegression()
clf.fit(X_resampled, y_resampled)
```

7.3 Under/Over-Sampling

SMOTEENN combines SMOTE with Edited Nearest Neighbours, which is used to pare down and centralise the negative cases.

```
from imblearn.combine import SMOTEENN

smo = SMOTEENN()
X_resampled, y_resampled = smo.fit_sample(X_train, y_train)
clf = LogisticRegression()
clf.fit(X_resampled, y_resampled)
```

7.4 Cost Sensitive Classification

One can also make the classifier aware of the imbalanced data by incorporating the weights of the classes into a cost function. Intuitively, we want to give higher weight to minority class and lower weight to majority class.

http://albahnsen.github.io/CostSensitiveClassification/index.html

CHAPTER 8

Data Leakage

Data leakage is a serious bane in machine learning, which usually results in overly optimistic model results.

8.1 Examples

Some subtle examples of data leakages.

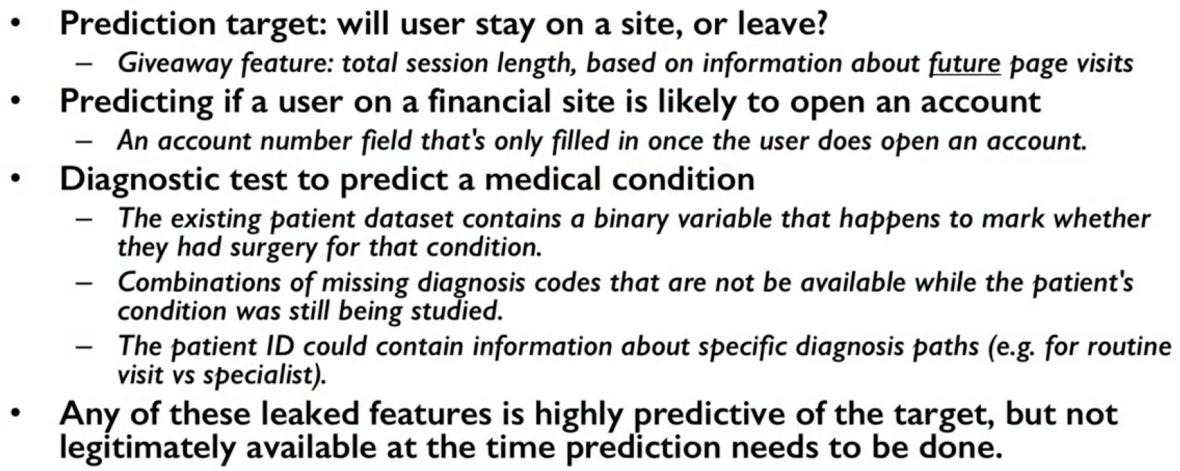


Fig. 1: University of Michigan: Coursera Data Science in Python

8.2 Types of Leakages

Data Leakages can be classified into two.

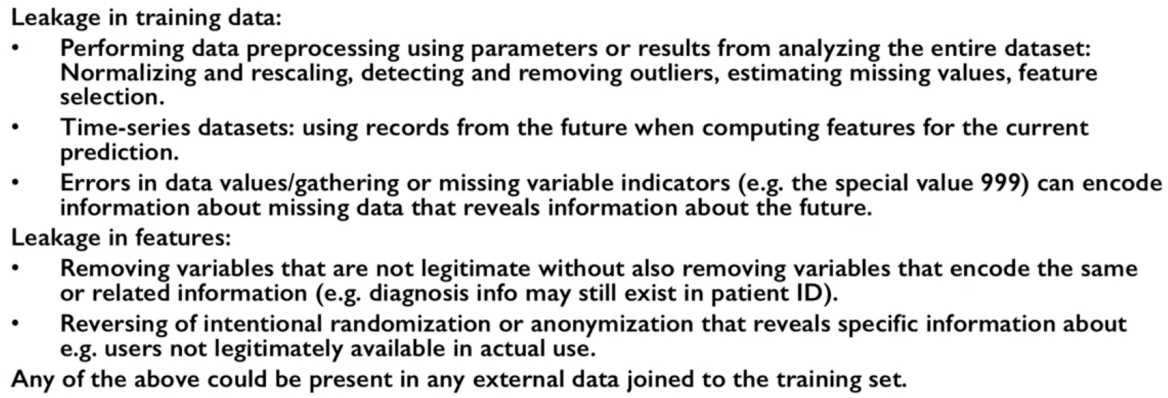


Fig. 2: University of Michigan: Coursera Data Science in Python

8.3 Detecting Leakages

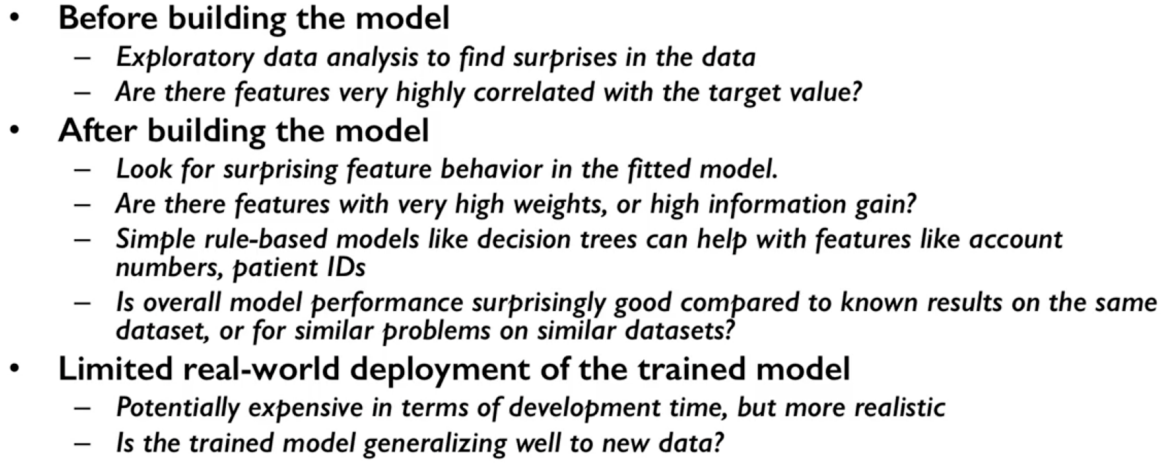


Fig. 3: University of Michigan: Coursera Data Science in Python

8.4 Minimising Leakages

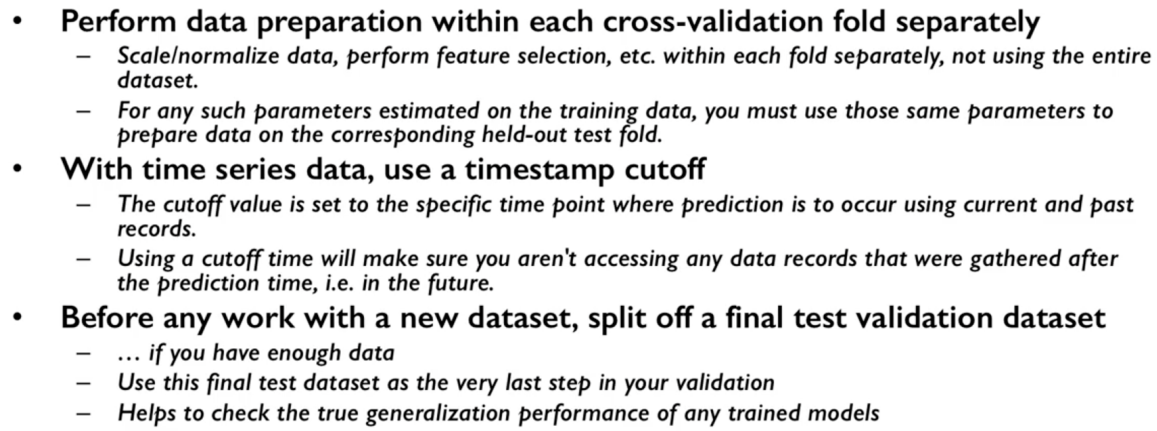


Fig. 4: University of Michigan: Coursera Data Science in Python

CHAPTER 9

Supervised Learning

9.1 Classification

When response is a categorical value.

9.1.1 K Nearest Neighbours (KNN)

Fig. 1: www.mathworks.com

Note:

1. **Distance Metric**: Eclidean Distance (default). In sklearn it is known as (Minkowski with p = 2)
2. **How many nearest neighbour**: k=1 very specific, k=5 more general model. Use nearest k data points to determine classification
3. **Weighting function on neighbours**: (optional)
4. **How to aggregate class of neighbour points**: Simple majority (default)

```
#### IMPORT MODULES ####
import pandas as pd
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.neighbors import KNeighborsClassifier



#### TRAIN TEST SPLIT ####
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

(continues on next page)

(continued from previous page)

```
#### CREATE MODEL ####
knn = KNeighborsClassifier(n_neighbors = 5)



#### FIT MODEL ####
knn.fit(X_train, y_train)
#KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
#      metric_params=None, n_jobs=1, n_neighbors=5, p=2,
#      weights='uniform')



#### TEST MODEL ####
knn.score(X_test, y_test)
0.53333333333333333
```

9.1.2 Decision Tree

Uses gini index (default) or entropy to split the data at binary level.

Strengths: Can select a large number of features that best determine the targets.

Weakness: Tends to overfit the data as it will split till the end. Pruning can be done to remove the leaves to prevent overfitting but that is not available in sklearn. Small changes in data can lead to different splits. Not very reproducible for future data (see random forest).

More more tuning parameters https://medium.com/@mohtedibf/indepth-parameter-tuning-for-decision-tree-6753118a03c3

```
###### IMPORT MODULES #######
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier



#### TRAIN TEST SPLIT ####
train_predictor, test_predictor, train_target, test_target = \
train_test_split(predictor, target, test_size=0.25)

print test_predictor.shape
print train_predictor.shape
(38, 4)
(112, 4)



#### CREATE MODEL ####
clf = DecisionTreeClassifier()



#### FIT MODEL ####
```

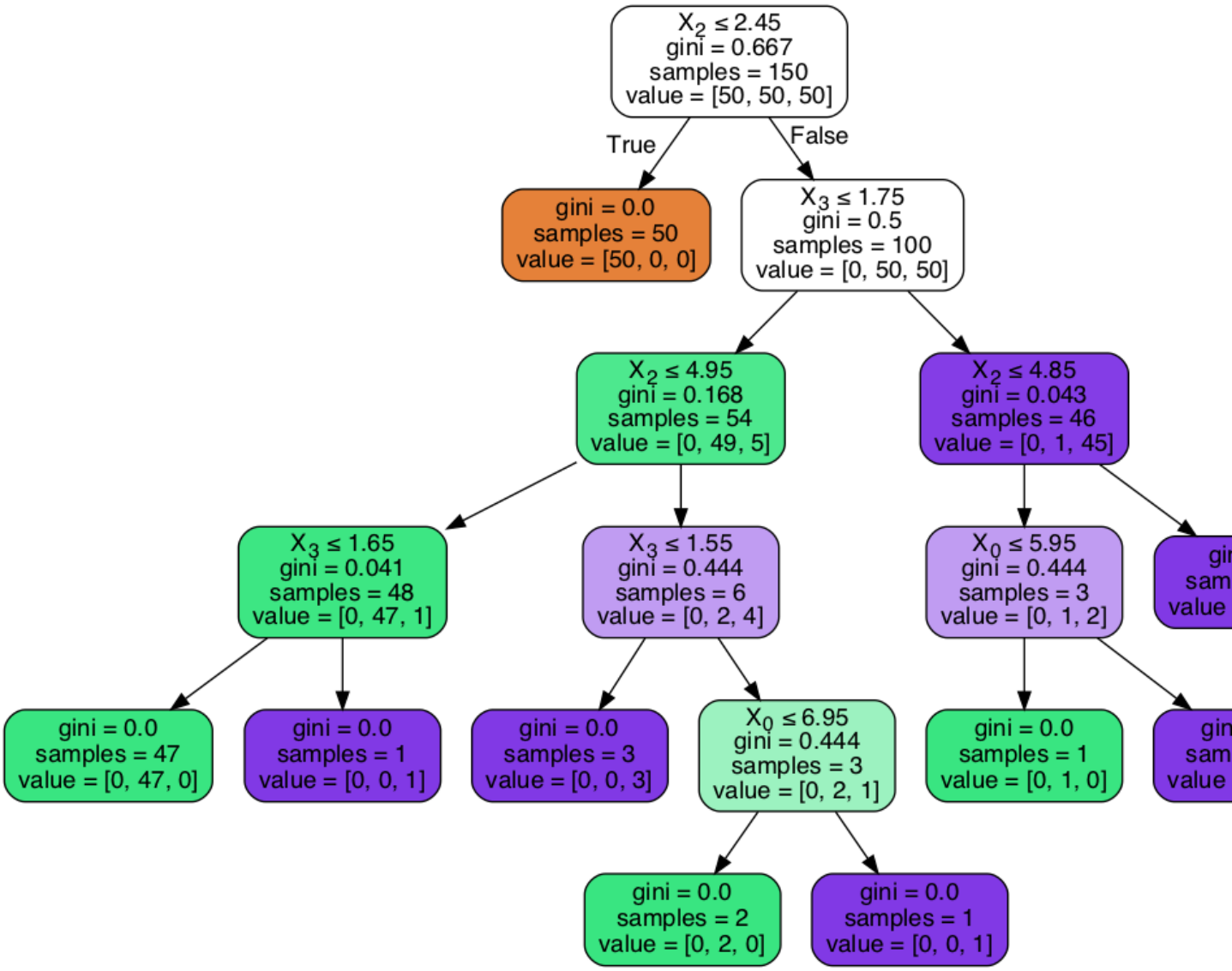
(continues on next page)

(continued from previous page)

```
model = clf.fit(train_predictor, train_target)
print model
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')



#### TEST MODEL ####
predictions = model.predict(test_predictor)

print sklearn.metrics.confusion_matrix(test_target,predictions)
print sklearn.metrics.accuracy_score(test_target, predictions)*100, '%'
[[14  0  0]
 [ 0 13  0]
 [ 0  1 10]]
97.3684210526 %


#### SCORE MODEL ####
# it is easier to use this package that does everything nicely for a perfect␣
↪confusion matrix
from pandas_confusion import ConfusionMatrix
ConfusionMatrix(test_target, predictions)
Predicted   setosa  versicolor  virginica  __all__
Actual
setosa          14           0          0       14
versicolor       0          13          0       13
virginica        0           1         10       11
__all__         14          14         10       38



####### FEATURE IMPORTANCE #### ####
f_impt= pd.DataFrame(model.feature_importances_, index=df.columns[:-2])
f_impt = f_impt.sort_values(by=0,ascending=False)
f_impt.columns = ['feature importance']
f_impt
petal width (cm)      0.952542
petal length (cm)     0.029591
sepal length (cm)     0.017867
sepal width (cm)      0.000000
```

Viewing the decision tree requires installing of the two packages *conda install graphviz* & *conda install pydotplus*.

```
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
dot_data = StringIO()
export_graphviz(dtree, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```


X2 ≤ 2.45
gini = 0.667
samples = 150
value = [50, 50, 50]
True
False
gini = 0.0
samples = 50
value = [50, 0, 0]
X3 ≤ 1.75
gini = 0.5
samples = 100
value = [0, 50, 50]
X2 ≤ 4.95
gini = 0.168
samples = 54
value = [0, 49, 5]
X2 ≤ 4.85
gini = 0.043
samples = 46
value = [0, 1, 45]
X3 ≤ 1.65
gini = 0.041
samples = 48
value = [0, 47, 1]
X3 ≤ 1.55
gini = 0.444
samples = 6
value = [0, 2, 4]
X0 ≤ 5.95
gini = 0.444
samples = 3
value = [0, 1, 2]
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]
X0 ≤ 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, 2, 0]
gini = 0.0
samples = 1
value = [0, 0, 1]

Parameters to tune decision trees include **maxdepth** & **min sample leaf**.

```
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_decision_tree
from adspy_shared_utilities import plot_feature_importances

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_state␣
↪= 0)

clf = DecisionTreeClassifier(max_depth = 4, min_samples_leaf = 8,
                             random_state = 0).fit(X_train, y_train)

plot_decision_tree(clf, cancer.feature_names, cancer.target_names)
```

9.1.3 Ensemble Learning

- Random Forests uses *bagging* (bootstrap aggregating) to implement ensemble learning
  - Many models are built by training on randomly-drawn subsets of the data
- *Boosting* is an alternate technique where each subsequent model in the ensemble boosts attributes that address data mis-classified by the previous model
- A *bucket of models* trains several different models using training data, and picks the one that works best with the test data
- *Stacking* runs multiple models at once on the data, and combines the results together
  - This is how the Netflix prize was won!

Fig. 2: Udemy Machine Learning Course by Frank Kane

9.1.4 Random Forest

An ensemble of decision trees.

- It is widely used and has very good results on many problems
- **sklearn.ensemble module**
  - Classification: `RandomForestClassifier`
  - Regression: `RandomForestRegressor`
- One decision tree tends to overfit
- Many decision trees tends to be more stable and generalised
- Ensemble of trees should be diverse: introduce random variation into tree building

Randomness is introduced by two ways:

- **Bootstrap**: AKA bagging. If your training set has N instances or samples in total, a bootstrap sample of size N is created by just repeatedly picking one of the N dataset rows at random with replacement, that

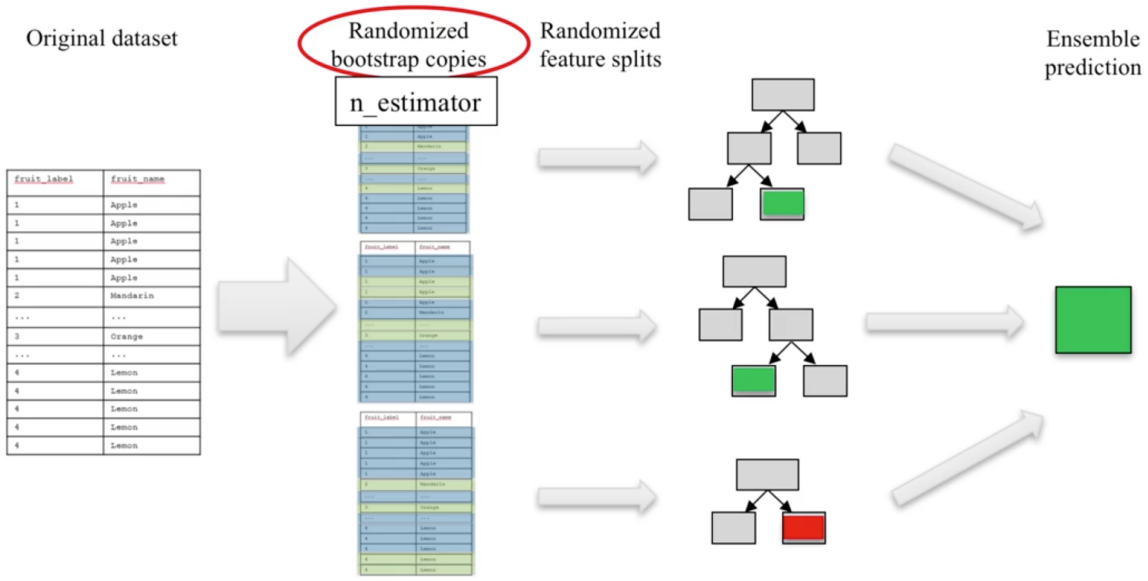


Fig. 3: University of Michigan: Coursera Data Science in Python

is, allowing for the possibility of picking the same row again at each selection. You repeat this random selection process N times. The resulting bootstrap sample has N rows just like the original training set but with possibly some rows from the original dataset missing and others occurring multiple times just due to the nature of the random selection with replacement. This process is repeated to generate n samples, using the parameter `n_estimators`, which will eventually generate n number decision trees.

- **Splitting Features**: When picking the best split for a node, instead of finding the best split across all possible features (decision tree), a random subset of features is chosen and the best split is found within that smaller subset of features. The number of features in the subset that are randomly considered at each stage is controlled by the `max_features` parameter.

This randomness in selecting the bootstrap sample to train an individual tree in a forest ensemble, combined with the fact that splitting a node in the tree is restricted to random subsets of the features of the split, virtually guarantees that all of the decision trees and the random forest will be different.

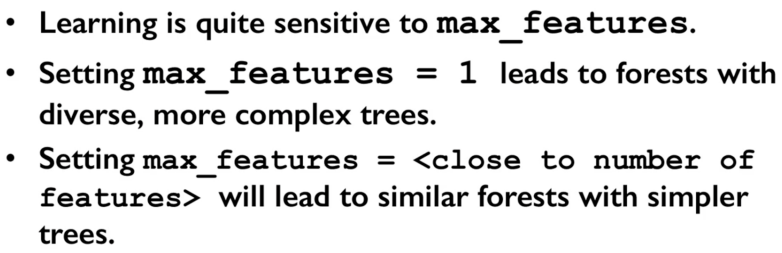


Fig. 4: University of Michigan: Coursera Data Science in Python

Prediction is then averaged among the trees.

Key parameters include `n_estimators`, `max_features`, `max_depth`, `n_jobs`.

```
###### IMPORT MODULES #### ###
import pandas as pd
import numpy as np
```

(continues on next page)

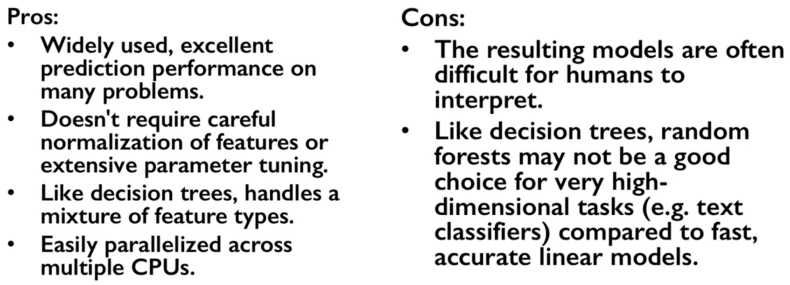


Fig. 5: University of Michigan: Coursera Data Science in Python

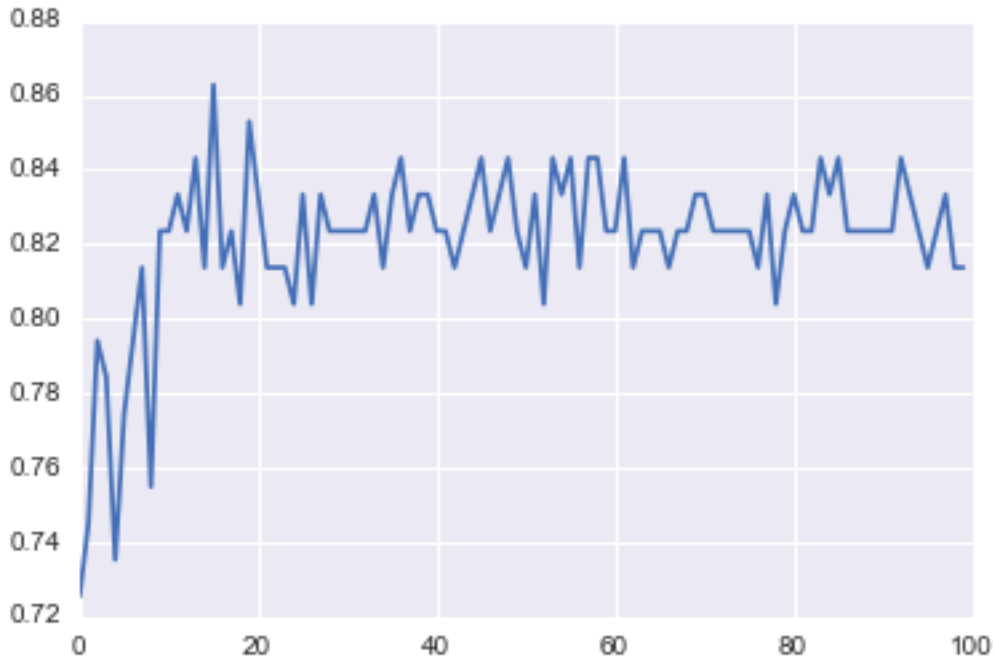
(continued from previous page)

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.cross_validation import train_test_split
import sklearn.metrics



#### TRAIN TEST SPLIT ####
train_feature, test_feature, train_target, test_target = \
train_test_split(feature, target, test_size=0.2)

print train_feature.shape
print test_feature.shape
(404, 13)
(102, 13)


#### CREATE MODEL ####
# use 100 decision trees
clf = RandomForestClassifier(n_estimators=100)



#### FIT MODEL ####
model = clf.fit(train_feature, train_target)
print model
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)



#### TEST MODEL ####
predictions = model.predict(test_feature)



#### SCORE MODEL ####
print 'accuracy', '\n', sklearn.metrics.accuracy_score(test_target, predictions)*100,
↪'%', '\n'
```

(continues on next page)

(continued from previous page)

```
print 'confusion matrix', '\n', sklearn.metrics.confusion_matrix(test_target,
↪predictions)
accuracy
82.3529411765 %
confusion matrix
[[21  0  3]
 [ 0 21  4]
 [ 8  3 42]]



####### FEATURE IMPORTANCE #### ####
# rank the importance of features
f_impt= pd.DataFrame(model.feature_importances_, index=df.columns[:-2])
f_impt = f_impt.sort_values(by=0,ascending=False)
f_impt.columns = ['feature importance']

RM    0.225612
LSTAT 0.192478
CRIM  0.108510
DIS   0.088056
AGE   0.074202
NOX   0.067718
B     0.057706
PTRATIO       0.051702
TAX   0.047568
INDUS 0.037871
RAD   0.026538
ZN    0.012635
CHAS  0.009405



#### GRAPHS ####

# see how many decision trees are minimally required make the accuarcy consistent
import numpy as np
import matplotlib.pylab as plt
import seaborn as sns
%matplotlib inline

trees=range(100)
accuracy=np.zeros(100)

for i in range(len(trees)):
  clf=RandomForestClassifier(n_estimators= i+1)
  model=clf.fit(train_feature, train_target)
  predictions=model.predict(test_feature)
  accuracy[i]=sklearn.metrics.accuracy_score(test_target, predictions)

plt.plot(trees,accuracy)

# well, seems like more than 10 trees will have a consistent accuracy of 0.82.
# Guess there's no need to have an ensemble of 100 trees!
```



9.1.5 Gradient Boosted Decision Trees

Gradient Boosted Decision Trees (GBDT) builds a series of small decision trees, with each tree attempting to correct errors from previous stage. Here's a good video on it, which describes AdaBoost, but gives a good overview of tree boosting models.

Typically, gradient boosted tree ensembles use lots of shallow trees known in machine learning as weak learners. Built in a nonrandom way, to create a model that makes fewer and fewer mistakes as more trees are added. Once the model is built, making predictions with a gradient boosted tree models is fast and doesn't use a lot of memory.

`learning_rate` parameter controls how hard each tree tries to correct mistakes from previous round. High learning rate, more complex trees.

Key parameters, `n_estimators`, `learning_rate`, `max_depth`.

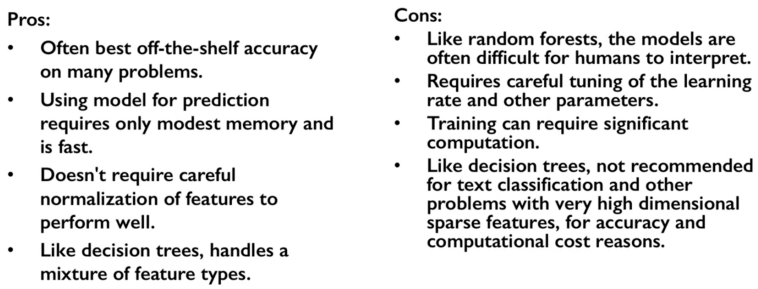


Fig. 6: University of Michigan: Coursera Data Science in Python

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_state␣
↪= 0)
```

(continues on next page)

(continued from previous page)

```
# Default Parameters
clf = GradientBoostingClassifier(random_state = 0)
clf.fit(X_train, y_train)

print('Breast cancer dataset (learning_rate=0.1, max_depth=3)')
print('Accuracy of GBDT classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of GBDT classifier on test set: {:.2f}\n'
     .format(clf.score(X_test, y_test)))

# Adjusting Learning Rate & Max Depth
clf = GradientBoostingClassifier(learning_rate = 0.01, max_depth = 2, random_state =␣
↪0)
clf.fit(X_train, y_train)

print('Breast cancer dataset (learning_rate=0.01, max_depth=2)')
print('Accuracy of GBDT classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of GBDT classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))


# Results
Breast cancer dataset (learning_rate=0.1, max_depth=3)
Accuracy of GBDT classifier on training set: 1.00
Accuracy of GBDT classifier on test set: 0.96

Breast cancer dataset (learning_rate=0.01, max_depth=2)
Accuracy of GBDT classifier on training set: 0.97
Accuracy of GBDT classifier on test set: 0.97
```

9.1.6 XGBoost

XGBoost or eXtreme Gradient Boosting, is a form of gradient boosted decision trees is that designed to be highly efficient, flexible and portable. It is one of the highly dominative classifier in competitive machine learning competitions.

https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/#

```
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(X, Y, random_state=0)

# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)

# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]

# evaluate predictions
```

(continues on next page)

(continued from previous page)

```
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

9.1.7 Light Gradient Boosting

LightGBM is a lightweight version of gradient boosting developed by Microsoft. It has similar performance to XGBoost but can run much faster than it.

https://lightgbm.readthedocs.io/en/latest/index.html

```
import lightgbm

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_
↪state=42, stratify=y)

# Create the LightGBM data containers
train_data = lightgbm.Dataset(X_train, label=y)
test_data = lightgbm.Dataset(X_test, label=y_test)

parameters = {
  'application': 'binary',
  'objective': 'binary',
  'metric': 'auc',
  'is_unbalance': 'true',
  'boosting': 'gbdt',
  'num_leaves': 31,
  'feature_fraction': 0.5,
  'bagging_fraction': 0.5,
  'bagging_freq': 20,
  'learning_rate': 0.05,
  'verbose': 0
}

model = lightgbm.train(parameters,
                       train_data,
                       valid_sets=test_data,
                       num_boost_round=5000,
                       early_stopping_rounds=100)
```

9.1.8 CatBoost

Category Boosting has high performance compared to other popular models, and does not require conversion of categorical values into numbers. It is also, like LightGBM, faster than XGBoost.

9.1.9 Naive Bayes

Naive Bayes is a probabilistic model. Features are assumed to be independent of each other in a given class. This makes the math very easy. E.g., words that are unrelated multiply together to form the final probability.

Prior Probability: Pr(y). Probability that a class (y) occurred in entire training dataset

Liklihood: Pr(y|xi) Probability of a class (y) occuring given all the features (xi).

There are 3 types of Naive Bayes:

- Bernouli: binary features (absence/presence of words)
- Multinomial: discrete features (account for frequency of words too, TF-IDF [frequency–inverse document frequency])
- Gaussian: continuous / real-value features (stores aerage avlue & standard deviation of each feature for each class)

Bernouli and Multinomial models are commonly used for sparse count data like text classification. The latter normally works better. Gaussian model is used for high-dimensional data.

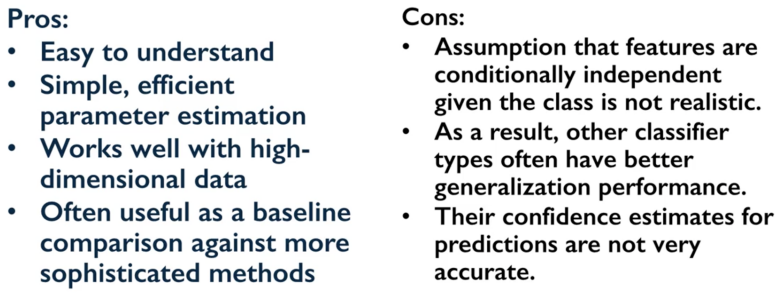


Fig. 7: University of Michigan: Coursera Data Science in Python

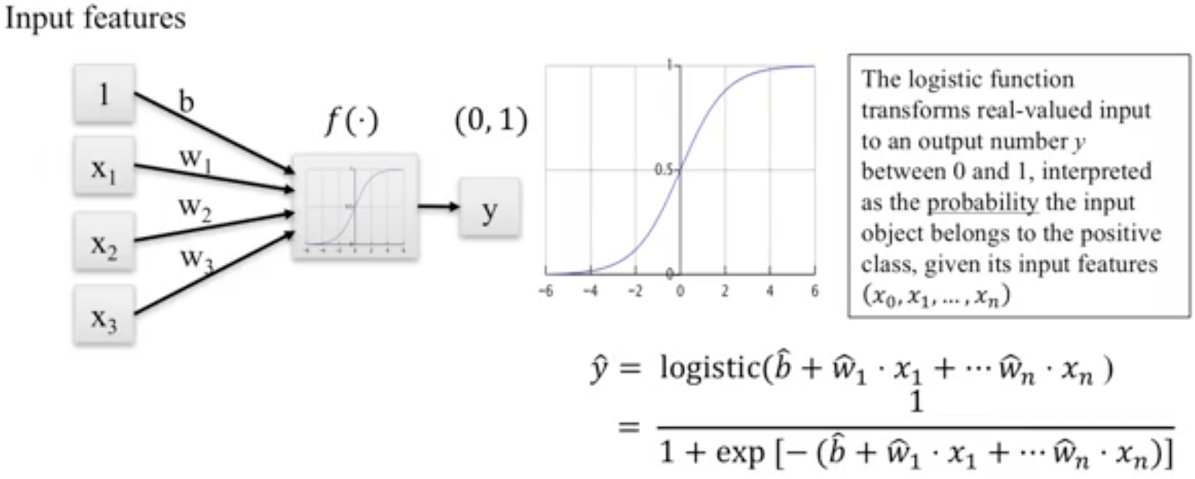
Sklearn allows **partial fitting**, i.e., fit the model incrementally if dataset is too large for memory.

Naive Bayes model only have one smoothing parameter called `alpha` (default 0.1). It adds a virtual data point that have positive values for all features. This is necessary considering that if there are no positive feature, the entire probability will be 0 (since it is a multiplicative model). More alpha means more smoothing, and more generalisation (less complex) model.

```
from sklearn.naive_bayes import GaussianNB
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2, random_state=0)

# no parameters for tuning
nbclf = GaussianNB().fit(X_train, y_train)
plot_class_regions_for_classifier(nbclf, X_train, y_train, X_test, y_test,
                                  'Gaussian Naive Bayes classifier: Dataset 1')

print('Accuracy of GaussianNB classifier on training set: {:.2f}'
     .format(nbclf.score(X_train, y_train)))
print('Accuracy of GaussianNB classifier on test set: {:.2f}'
     .format(nbclf.score(X_test, y_test)))
```

9.1.10 Logistic Regression

Binary output or y value. Functions are available in both statsmodels and sklearn packages.




```
#### IMPORT MODULES ####
import pandas as pd
import statsmodels.api as sm



#### FIT MODEL ####
lreg = sm.Logit(df3['diameter_cut'], df3[trainC]).fit()
print lreg.summary()



Optimization terminated successfully.
       Current function value: 0.518121
       Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:           diameter_cut   No. Observations:                18067
Model:                          Logit   Df Residuals:                    18065
Method:                           MLE   Df Model:                            1
Date:                Thu, 04 Aug 2016   Pseudo R-squ.:                0.2525
Time:                        14:13:14   Log-Likelihood:               -9360.9
converged:                       True   LL-Null:                      -12523.
                                        LLR p-value:                     0.000
==============================================================================
                    coef    std err          z      P>|z|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
depth             4.2529      0.067     63.250      0.000         4.121     4.385
layers_YESNO     -2.1102      0.037    -57.679      0.000        -2.182    -2.039
==============================================================================



#### CONFIDENCE INTERVALS ####
params = lreg.params
conf = lreg.conf_int()
conf['OR'] = params
conf.columns = ['Lower CI', 'Upper CI', 'OR']
print (np.exp(conf))
```

(continues on next page)

(continued from previous page)

```
Lower CI   Upper CI         OR
depth         61.625434  80.209893  70.306255
layers_YESNO   0.112824   0.130223   0.121212
```

A regularisation penlty L2, just like ridge regression is by default in `sklearn.linear_model`, `LogisticRegression`, controlled using the parameter C (default 1).

```
from sklearn.linear_model import LogisticRegression
from adspy_shared_utilities import (plot_class_regions_for_classifier_subplot)

fig, subaxes = plt.subplots(1, 1, figsize=(7, 5))
y_fruits_apple = y_fruits_2d == 1   # make into a binary problem: apples vs␣
↪everything else
X_train, X_test, y_train, y_test = (
train_test_split(X_fruits_2d.as_matrix(),
                 y_fruits_apple.as_matrix(),
                 random_state = 0))

clf = LogisticRegression(C=100).fit(X_train, y_train)
plot_class_regions_for_classifier_subplot(clf, X_train, y_train, None,
                                          None, 'Logistic regression \
for binary classification\nFruit dataset: Apple vs others',
                                          subaxes)

h = 6
w = 8
print('A fruit with height {} and width {} is predicted to be: {}'
     .format(h,w, ['not an apple', 'an apple'][clf.predict([[h,w]])[0]]))

h = 10
w = 7
print('A fruit with height {} and width {} is predicted to be: {}'
     .format(h,w, ['not an apple', 'an apple'][clf.predict([[h,w]])[0]]))
subaxes.set_xlabel('height')
subaxes.set_ylabel('width')

print('Accuracy of Logistic regression classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Logistic regression classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

9.1.11 Support Vector Machine

Support Vector Machines (SVM) involves locating the support vectors of two boundaries to find a maximum tolerance hyperplane. Side note: linear kernels work best for text classification.

Have 3 tuning parameters. Need to normalize first too!

1. Have regularisation using parameter C, just like logistic regression. Default to 1. Limits the importance of each point.
2. Type of kernel. Default is Radial Basis Function (RBF)
3. Gamma parameter for adjusting kernel width. Influence of a single training example reaches. Low gamma > far reach, high values > limited reach.

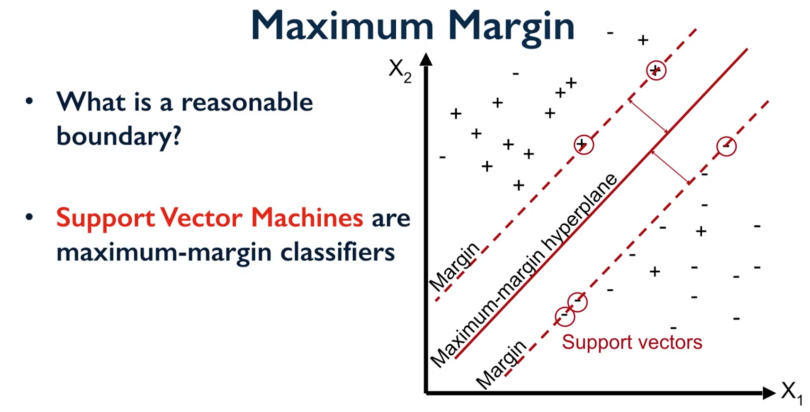
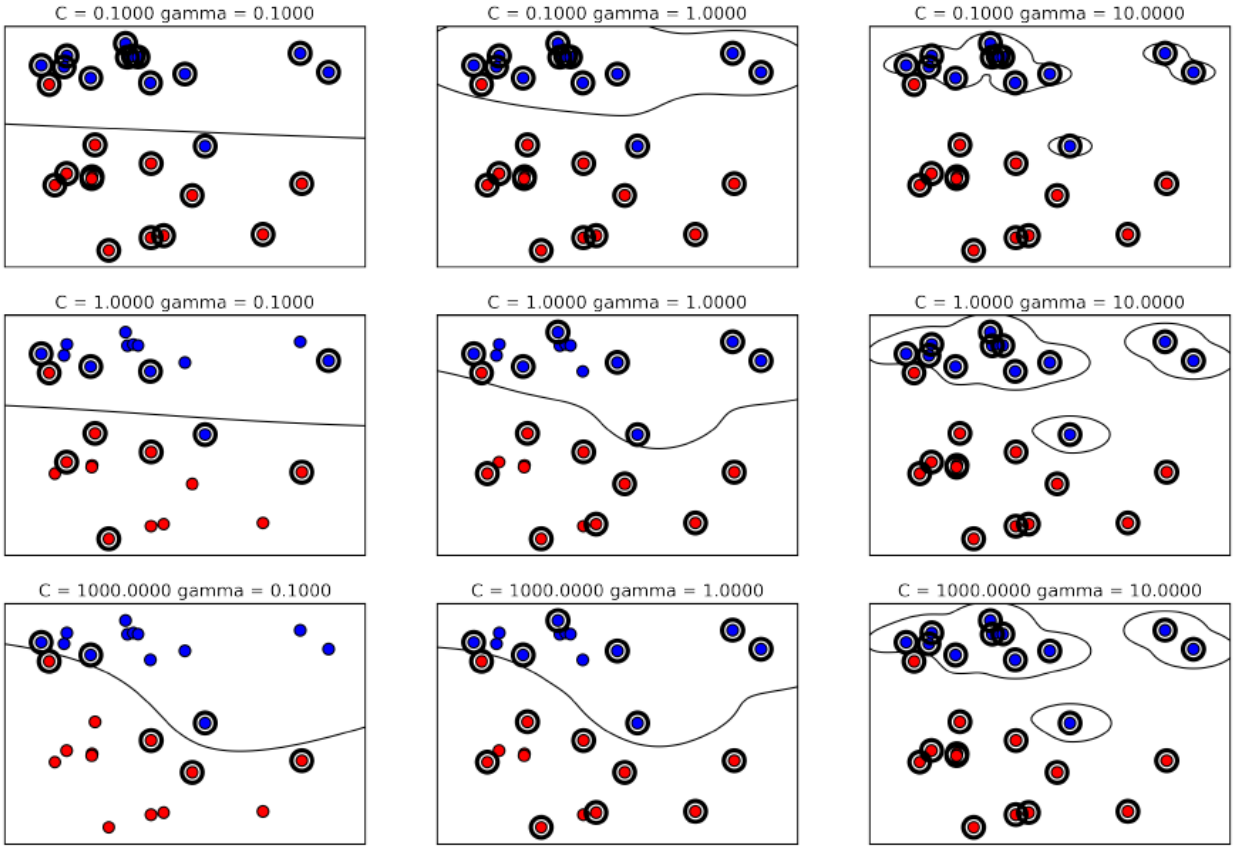


Fig. 8: University of Michigan: Coursera Data Science in Python




```
from sklearn.svm import SVC
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot


X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2, random_state = 0)

fig, subaxes = plt.subplots(1, 1, figsize=(7, 5))
this_C = 1.0
clf = SVC(kernel = 'linear', C=this_C).fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
title = 'Linear SVC, C = {:.3f}'.format(this_C)
plot_class_regions_for_classifier_subplot(clf, X_train, y_train, None, None, title,␣
↪subaxes)
```

We can directly call a linear SVC by directly importing the `LinearSVC` function

```
from sklearn.svm import LinearSVC
X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_state␣
↪= 0)

clf = LinearSVC().fit(X_train, y_train)
print('Breast cancer dataset')
print('Accuracy of Linear SVC classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of Linear SVC classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

Multi-Class Classification, i.e., having more than 2 target values, is also possible. With the results, it is possible to compare one class versus all other classes.

```
from sklearn.svm import LinearSVC

X_train, X_test, y_train, y_test = train_test_split(X_fruits_2d, y_fruits_2d, random_
↪state = 0)

clf = LinearSVC(C=5, random_state = 67).fit(X_train, y_train)
print('Coefficients:\n', clf.coef_)
print('Intercepts:\n', clf.intercept_)
```

visualising in a graph...

```
plt.figure(figsize=(6,6))
colors = ['r', 'g', 'b', 'y']
cmap_fruits = ListedColormap(['#FF0000', '#00FF00', '#0000FF','#FFFF00'])

plt.scatter(X_fruits_2d[['height']], X_fruits_2d[['width']],
            c=y_fruits_2d, cmap=cmap_fruits, edgecolor = 'black', alpha=.7)

x_0_range = np.linspace(-10, 15)

for w, b, color in zip(clf.coef_, clf.intercept_, ['r', 'g', 'b', 'y']):
    # Since class prediction with a linear model uses the formula y = w_0 x_0 + w_1 x_
↪1 + b,
    # and the decision boundary is defined as being all points with y = 0, to plot x_
↪1 as a
    # function of x_0 we just solve w_0 x_0 + w_1 x_1 + b = 0 for x_1:
    plt.plot(x_0_range, -(x_0_range * w[0] + b) / w[1], c=color, alpha=.8)

plt.legend(target_names_fruits)
plt.xlabel('height')
plt.ylabel('width')
plt.xlim(-2, 12)
plt.ylim(-2, 15)
plt.show()
```

Kernalised Support Vector Machines

For complex classification, new dimensions can be added to SVM. e.g., square of x. There are many types of kernal transformations. By default, SVM will use the Radial Basis Function (RBF) kernel.

```
from sklearn.svm import SVC
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0)

# The default SVC kernel is radial basis function (RBF)
plot_class_regions_for_classifier(SVC().fit(X_train, y_train),
                                  X_train, y_train, None, None,
                                  'Support Vector Classifier: RBF kernel')

# Compare decision boundries with polynomial kernel, degree = 3
plot_class_regions_for_classifier(SVC(kernel = 'poly', degree = 3)
                                  .fit(X_train, y_train), X_train,
                                  y_train, None, None,
                                  'Support Vector Classifier: Polynomial kernel,␣
↪degree = 3')
```

Full tuning in Support Vector Machines, using normalisation, kernel tuning, and regularisation.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

clf = SVC(kernel = 'rbf', gamma=1, C=10).fit(X_train_scaled, y_train)
print('Breast cancer dataset (normalized with MinMax scaling)')
print('RBF-kernel SVC (with MinMax scaling) training set accuracy: {:.2f}'
     .format(clf.score(X_train_scaled, y_train)))
print('RBF-kernel SVC (with MinMax scaling) test set accuracy: {:.2f}'
     .format(clf.score(X_test_scaled, y_test)))
```

9.1.12 Neural Networks

Examples using Multi-Layer Perceptrons (MLP).

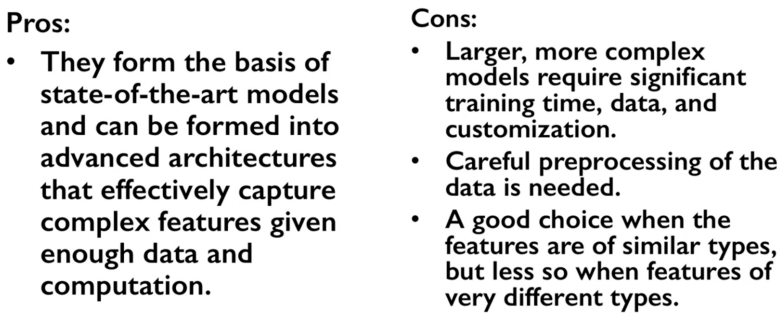


Fig. 9: University of Michigan: Coursera Data Science in Python

Parameters include

- `hidden_layer_sizes` which is the number of hidden layers, with no. units in each layer (default 100).

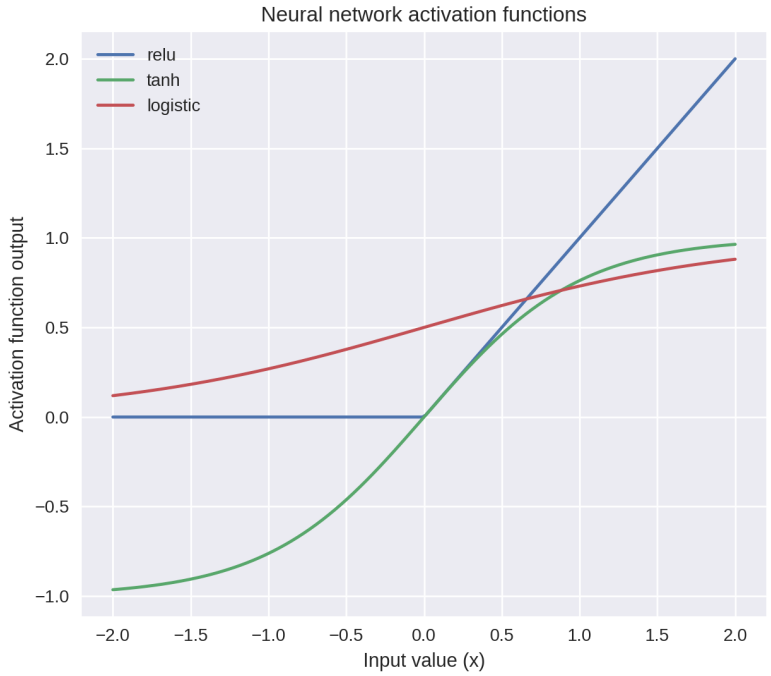


Fig. 10: Activation Function. University of Michigan: Coursera Data Science in Python

- `solvers` is the algorithm usedthat does the numerical work of finding the optimal weights. default `adam` used for large datasets, `lbfgs` is used for smaller datasets.
- `alpha`: L2 regularisation, default is 0.0001,
- `activation`: non-linear function used for activation function which include `relu` (default), `logistic`, `tanh`

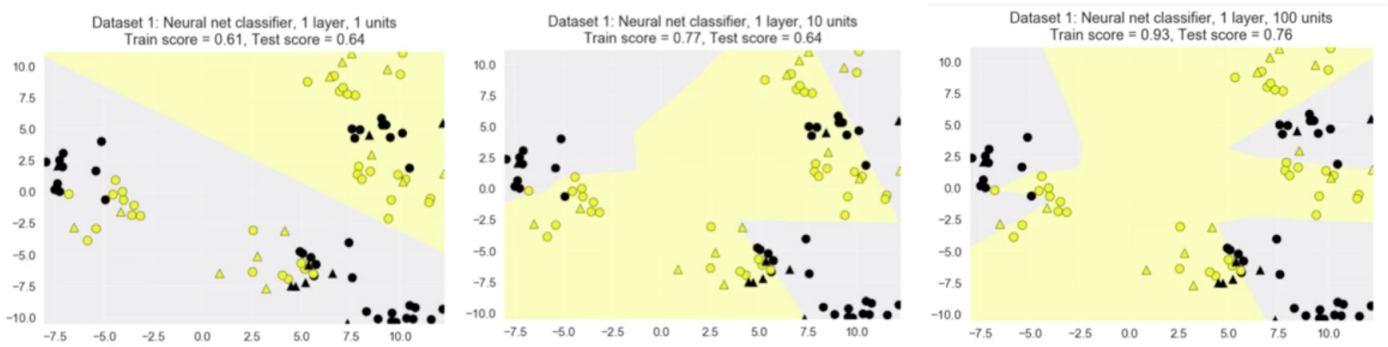
One Hidden Layer

```
from sklearn.neural_network import MLPClassifier
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state=0)

fig, subaxes = plt.subplots(3, 1, figsize=(6,18))


for units, axis in zip([1, 10, 100], subaxes):
    nnclf = MLPClassifier(hidden_layer_sizes = [units], solver='lbfgs',
                          random_state = 0).fit(X_train, y_train)

    title = 'Dataset 1: Neural net classifier, 1 layer, {} units'.format(units)

    plot_class_regions_for_classifier_subplot(nnclf, X_train, y_train,
                                              X_test, y_test, title, axis)
    plt.tight_layout()
```

Two Hidden Layers, L2 Regularisation (alpha), Activation

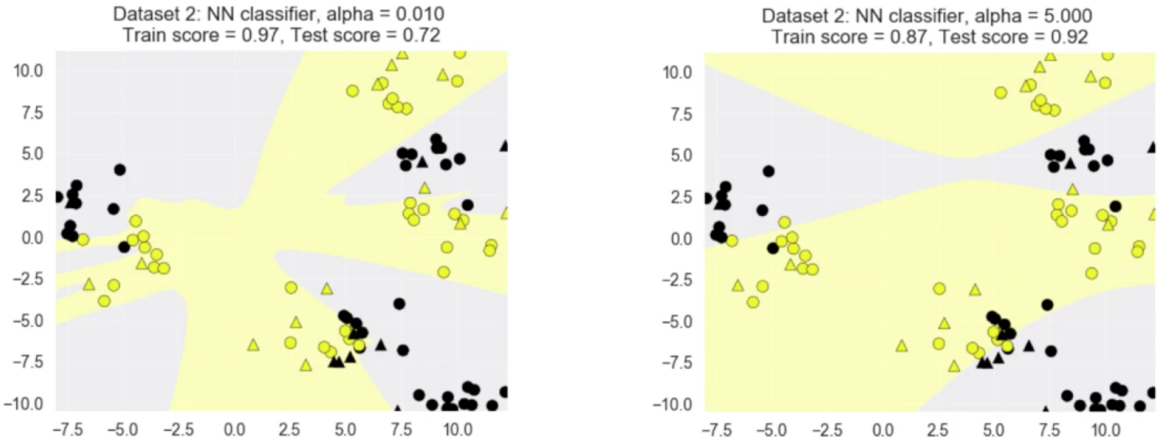
```
X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state=0)
```

(continues on next page)



(continued from previous page)

```
fig, subaxes = plt.subplots(4, 1, figsize=(6, 23))

for this_alpha, axis in zip([0.01, 0.1, 1.0, 5.0], subaxes):
    nnclf = MLPClassifier(solver='lbfgs', activation = 'tanh',
                          alpha = this_alpha,
                          hidden_layer_sizes = [100, 100],
                          random_state = 0).fit(X_train, y_train)

    title = 'Dataset 2: NN classifier, alpha = {:.3f} '.format(this_alpha)

    plot_class_regions_for_classifier_subplot(nnclf, X_train, y_train,
                                              X_test, y_test, title, axis)
    plt.tight_layout()
```



alpha = 0.01 alpha = 5.0

Normalisation: Input features should be normalised.

```
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler


scaler = MinMaxScaler()
```

(continues on next page)

(continued from previous page)

```
X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_state␣
↪= 0)
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

clf = MLPClassifier(hidden_layer_sizes = [100, 100], alpha = 5.0,
                    random_state = 0, solver='lbfgs').fit(X_train_scaled, y_train)

print('Breast cancer dataset')
print('Accuracy of NN classifier on training set: {:.2f}'
     .format(clf.score(X_train_scaled, y_train)))
print('Accuracy of NN classifier on test set: {:.2f}'
     .format(clf.score(X_test_scaled, y_test)))


# RESULTS
Breast cancer dataset
Accuracy of NN classifier on training set: 0.98
Accuracy of NN classifier on test set: 0.97
```

9.1.13 Auto-ML

Quite a number of open-sourced automatic machine learning packages have been released. It selects the best models and their hyperparameters, making it extremely easy to train supervised models.

Auto Keras

Uses neural network for training. Similar to Google's AutoML approach.

```
import autokeras as ak

clf = ak.ImageClassifier()
clf.fit(x_train, y_train)
results = clf.predict(x_test)
```

Auto Sklearn

Using Bayesian optimizer, this automation trains using models within sklearn.

```
import autosklearn.classification
import sklearn.model_selection
import sklearn.datasets
import sklearn.metrics

X, y = sklearn.datasets.load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = \
        sklearn.model_selection.train_test_split(X, y, random_state=1)

# time_left_for_this_task (total) must be more than per_run_time_limit
automl = autosklearn.classification.AutoSklearnClassifier(time_left_for_this_task=60,
                                                           per_run_time_limit=30)
```

(continues on next page)

(continued from previous page)

```
automl.fit(X_train, y_train)
y_hat = automl.predict(X_test)
print("Accuracy score", sklearn.metrics.accuracy_score(y_test, y_hat))
```

We can see the total results

```
print(automl.sprint_statistics())

# auto-sklearn results:
#   Dataset name: cb9e4c17575300e0bcdb85c0920f3385
#   Metric: accuracy
#   Best validation score: 1.000000
#   Number of target algorithm runs: 51
#   Number of successful target algorithm runs: 47
#   Number of crashed target algorithm runs: 3
#   Number of target algorithms that exceeded the time limit: 1
#   Number of target algorithms that exceeded the memory limit: 0
```

Auto WEKA

WEKA is a GUI-based software for easy quick analysis of datasets. It is the same in concept as Auto Sklearn but have a wider range of models and hyperparameters.

9.2 Regression

When response is a continuous value.

9.2.1 OLS Regression

Ordinary Least Squares Regression or OLS Regression is the most basic form and fundamental of regression. Best fit line $\hat{y} = a + bx$ is drawn based on the ordinary least squares method. i.e., least total area of squares (sum of squares) with length from each x,y point to regresson line.

OLS can be conducted using statsmodel package.

```
model = smf.ols(formula='diameter ~ depth', data=df3).fit()
print model.summary()



OLS Regression Results
==============================================================================
Dep. Variable:               diameter   R-squared:                       0.512
Model:                            OLS   Adj. R-squared:                  0.512
Method:                 Least Squares   F-statistic:                 1.895e+04
Date:                Tue, 02 Aug 2016   Prob (F-statistic):               0.00
Time:                        17:10:34   Log-Likelihood:                -51812.
No. Observations:               18067   AIC:                         1.036e+05
```

(continues on next page)

(continued from previous page)

```
Df Residuals:                    18065   BIC:                          1.036e+05
Df Model:                            1
Covariance Type:            nonrobust
==============================================================================
coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      2.2523      0.054     41.656      0.000         2.146     2.358
depth         11.5836      0.084    137.675      0.000        11.419    11.749
==============================================================================
Omnibus:                    12117.030   Durbin-Watson:                   0.673
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           391356.565
Skew:                           2.771   Prob(JB):                         0.00
Kurtosis:                      25.117   Cond. No.                         3.46
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

or sci-kit learn package

```
from sklearn import linear_model

reg = linear_model.LinearRegression()
model = reg.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])

model
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
reg.coef_
array([ 0.5,  0.5])

# R2 scores
r2_trains = model.score(X_train, y_train)
r2_tests = model.score(X_test, y_test)
```

9.2.2 Ridge Regression

Regularisaton is an important concept used in Ridge Regression as well as the next LASSO regression. Ridge regression uses regularisation which adds a penalty parameter to a variable when it has a large variation. Regularisation prevents overfitting by restricting the model, thus lowering its complexity.

- Uses L2 regularisation, which *reduces the sum of squares* of the parameters
- The influence of L2 is controlled by an alpha parameter. Default is 1.
- High alpha means more regularisation and a simpler model.
- More in https://www.analyticsvidhya.com/blog/2016/01/complete-tutorial-ridge-lasso-regression-python/

```
#### IMPORT MODULES ####
import panda as pd
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.preprocessing import MinMaxScaler

#### TRAIN-TEST SPLIT ####
```

(continues on next page)

(continued from previous page)

```
X_train, X_test, y_train, y_test = train_test_split(X_crime, y_crime,
                                                    random_state = 0)

#### NORMALIZATION ####
  # using minmaxscaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


#### CREATE AND FIT MODEL ####
linridge = Ridge(alpha=20.0).fit(X_train_scaled, y_train)

print('Crime dataset')
print('ridge regression linear model intercept: {}'
   .format(linridge.intercept_))
print('ridge regression linear model coeff:\n{}'
   .format(linridge.coef_))
print('R-squared score (training): {:.3f}'
   .format(linridge.score(X_train_scaled, y_train)))
print('R-squared score (test): {:.3f}'
   .format(linridge.score(X_test_scaled, y_test)))
print('Number of non-zero features: {}'
   .format(np.sum(linridge.coef_ != 0)))
```

To investigate the effect of alpha:

```
print('Ridge regression: effect of alpha regularization parameter\n')
for this_alpha in [0, 1, 10, 20, 50, 100, 1000]:
  linridge = Ridge(alpha = this_alpha).fit(X_train_scaled, y_train)
  r2_train = linridge.score(X_train_scaled, y_train)
  r2_test = linridge.score(X_test_scaled, y_test)
  num_coeff_bigger = np.sum(abs(linridge.coef_) > 1.0)
  print('Alpha = {:.2f}\nnum abs(coeff) > 1.0: {}, \
        r-squared training: {:.2f}, r-squared test: {:.2f}\n'
       .format(this_alpha, num_coeff_bigger, r2_train, r2_test))
```

Note:

- Many variables with small/medium effects: Ridge
- Only a few variables with medium/large effects: LASSO

9.2.3 LASSO Regression

LASSO refers to Least Absolute Shrinkage and Selection Operator Regression. Like Ridge Regression this also has a regularisation property.

- Uses L1 regularisation, which *reduces sum of the absolute values of coefficients*, that change unimportant features (their regression coefficients) into 0
- This is known as a sparse solution, or a kind of feature selection, since some variables were removed in the process
- The influence of L1 is controlled by an alpha parameter. Default is 1.

- High alpha means more regularisation and a simpler model. When alpha = 0, then it is a normal OLS regression.

a. Bias increase & variability decreases when alpha increases.

b. Useful when there are many features (explanatory variables).

c. Have to standardize all features so that they have mean 0 and std error 1.

d. Have several algorithms: LAR (Least Angle Regression). Starts w 0 predictors & add each predictor that is most correlated at each step.

```
#### IMPORT MODULES ####
import pandas as pd
import numpy as py
from sklearn import preprocessing
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LassoLarsCV
import sklearn.metrics
from sklearn.datasets import load_boston



#### NORMALIZATION ####
# standardise the means to 0 and standard error to 1
for i in df.columns[:-1]: # df.columns[:-1] = dataframe for all features
  df[i] = preprocessing.scale(df[i].astype('float64'))
df.describe()



#### TRAIN TEST SPLIT ####
train_feature, test_feature, train_target, test_target = \
train_test_split(feature, target, random_state=123, test_size=0.2)

print train_feature.shape
print test_feature.shape
(404, 13)
(102, 13)



#### CREATE MODEL ####
# Fit the LASSO LAR regression model
# cv=10; use k-fold cross validation
# precompute; True=model will be faster if dataset is large
model=LassoLarsCV(cv=10, precompute=False)



#### FIT MODEL ####
model = model.fit(train_feature,train_target)
print model
LassoLarsCV(copy_X=True, cv=10, eps=2.2204460492503131e-16,
      fit_intercept=True, max_iter=500, max_n_alphas=1000, n_jobs=1,
      normalize=True, positive=False, precompute=False, verbose=False)



#### ANALYSE COEFFICIENTS ####
```

(continues on next page)

(continued from previous page)

```
Compare the regression coefficients, and see which one LASSO removed.
LSTAT is the most important predictor, followed by RM, DIS, and RAD. AGE is removed␣
↪by LASSO

df2=pd.DataFrame(model.coef_, index=feature.columns)
df2.sort_values(by=0,ascending=False)
RM     3.050843
RAD    2.040252
ZN     1.004318
B      0.629933
CHAS   0.317948
INDUS  0.225688
AGE    0.000000
CRIM   -0.770291
NOX    -1.617137
TAX    -1.731576
PTRATIO        -1.923485
DIS    -2.733660
LSTAT  -3.878356



#### SCORE MODEL ####
# MSE from training and test data
from sklearn.metrics import mean_squared_error
train_error = mean_squared_error(train_target, model.predict(train_feature))
test_error = mean_squared_error(test_target, model.predict(test_feature))

print ('training data MSE')
print(train_error)
print ('test data MSE')
print(test_error)

# MSE closer to 0 are better
# test dataset is less accurate as expected
training data MSE
20.7279948891
test data MSE
28.3767672242


# R-square from training and test data
rsquared_train=model.score(train_feature,train_target)
rsquared_test=model.score(test_feature,test_target)
print ('training data R-square')
print(rsquared_train)
print ('test data R-square')
print(rsquared_test)

# test data explained 65% of the predictors
training data R-square
0.755337444405
test data R-square
0.657019301268
```

9.2.4 Polynomial Regression

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures


# Normal Linear Regression
X_train, X_test, y_train, y_test = train_test_split(X_F1, y_F1,
                                                    random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)

print('linear model coeff (w): {}'
     .format(linreg.coef_))
print('linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('R-squared score (test): {:.3f}'
     .format(linreg.score(X_test, y_test)))

print('\nNow we transform the original input data to add\n\
polynomial features up to degree 2 (quadratic)\n')

# Polynomial Regression
poly = PolynomialFeatures(degree=2)
X_F1_poly = poly.fit_transform(X_F1)

X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, y_F1,
                                                    random_state = 0)
linreg = LinearRegression().fit(X_train, y_train)

print('(poly deg 2) linear model coeff (w):\n{}'
     .format(linreg.coef_))
print('(poly deg 2) linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('(poly deg 2) R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
print('(poly deg 2) R-squared score (test): {:.3f}\n'
     .format(linreg.score(X_test, y_test)))

# Polynomial with Ridge Regression
'''Addition of many polynomial features often leads to
overfitting, so we often use polynomial features in combination
with regression that has a regularization penalty, like ridge
regression.'''

X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, y_F1,
                                                    random_state = 0)
linreg = Ridge().fit(X_train, y_train)

print('(poly deg 2 + ridge) linear model coeff (w):\n{}'
     .format(linreg.coef_))
print('(poly deg 2 + ridge) linear model intercept (b): {:.3f}'
     .format(linreg.intercept_))
print('(poly deg 2 + ridge) R-squared score (training): {:.3f}'
     .format(linreg.score(X_train, y_train)))
```

(continues on next page)

(continued from previous page)

```
print('(poly deg 2 + ridge) R-squared score (test): {:.3f}'
      .format(linreg.score(X_test, y_test)))
```

9.2.5 Decision Tree Regressor

Same as decision tree classifier but the target is continuous.

```
from sklearn.tree import DecisionTreeRegressor
```

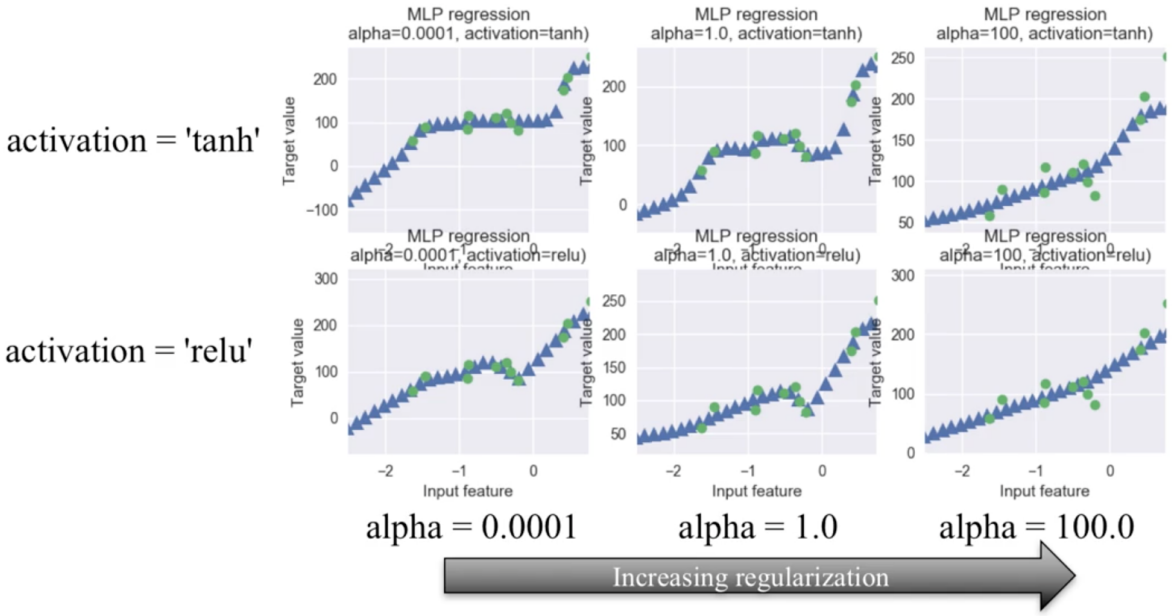
9.2.6 Random Forest Regressor

Same as randomforest classifier but the target is continuous.

```
from sklearn.ensemble import RandomForestRegressor
```

9.2.7 Neutral Networks

```
from sklearn.neural_network import MLPRegressor

fig, subaxes = plt.subplots(2, 3, figsize=(11,8), dpi=70)

X_predict_input = np.linspace(-3, 3, 50).reshape(-1,1)

X_train, X_test, y_train, y_test = train_test_split(X_R1[0::5], y_R1[0::5], random_
↪state = 0)

for thisaxisrow, thisactivation in zip(subaxes, ['tanh', 'relu']):
    for thisalpha, thisaxis in zip([0.0001, 1.0, 100], thisaxisrow):
        mlpreg = MLPRegressor(hidden_layer_sizes = [100,100],
                              activation = thisactivation,
                              alpha = thisalpha,
                              solver = 'lbfgs').fit(X_train, y_train)
        y_predict_output = mlpreg.predict(X_predict_input)
        thisaxis.set_xlim([-2.5, 0.75])
        thisaxis.plot(X_predict_input, y_predict_output,
                      '^', markersize = 10)
        thisaxis.plot(X_train, y_train, 'o')
        thisaxis.set_xlabel('Input feature')
        thisaxis.set_ylabel('Target value')
        thisaxis.set_title('MLP regression\nalpha={}, activation={})'
                           .format(thisalpha, thisactivation))
        plt.tight_layout()
```


activation = 'tanh'
activation = 'relu'
MLP regression
alpha=0.0001, activation=tanh)
MLP regression
alpha=1.0, activation=tanh)
MLP regression
alpha=100, activation=tanh)
MLP regression
alpha=0.0001, activation=relu)
MLP regression
alpha=1.0, activation=relu)
MLP regression
alpha=100, activation=relu)
Target value
Input feature
alpha = 0.0001
alpha = 1.0
alpha = 100.0
Increasing regularization

CHAPTER 10

Unsupervised Learning

No labeled responses, the goal is to capture interesting structure or information.

Applications include:

- Visualise structure of a complex dataset
- Density estimations to predict probabilities of events
- Compress and summarise the data
- Extract features for supervised learning
- Discover important clusters or outliers

10.1 Transformations

Processes that extract or compute information.

10.1.1 Kernel Density Estimation

10.1.2 Dimensionality Reduction

- **Curse of Dimensionality**: Very hard to visualise with many dimensions
- Finds an approximate version of your dataset using fewer features
- Used for exploring and visualizing a dataset to understand grouping or relationships
- Often visualized using a 2-dimensional scatterplot
- Also used for compression, finding features for supervised learning
- Can be classified into linear (PCA), or non-linear (manifold) reduction techniques

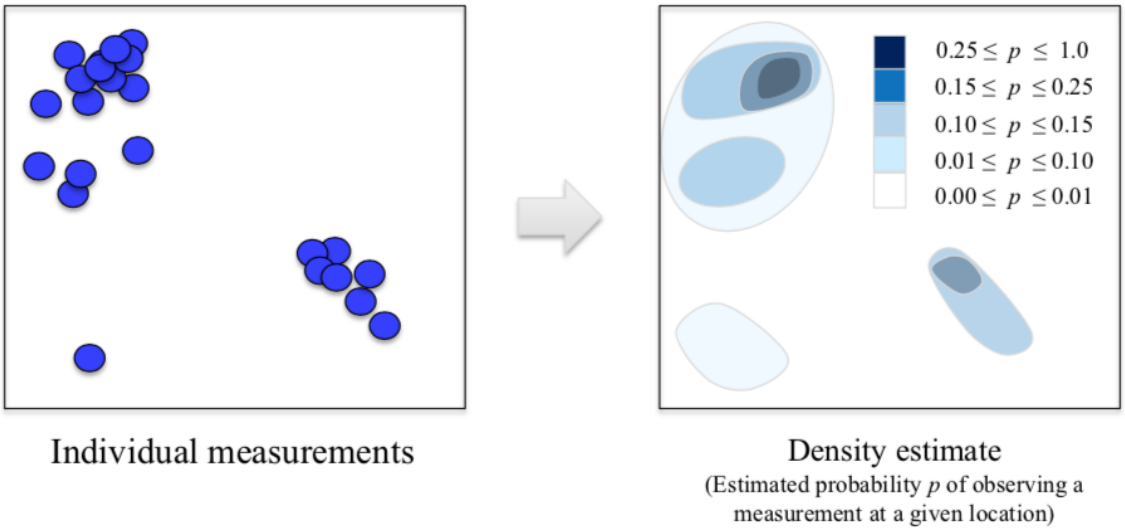


Fig. 1: University of Michigan: Coursera Data Science in Python

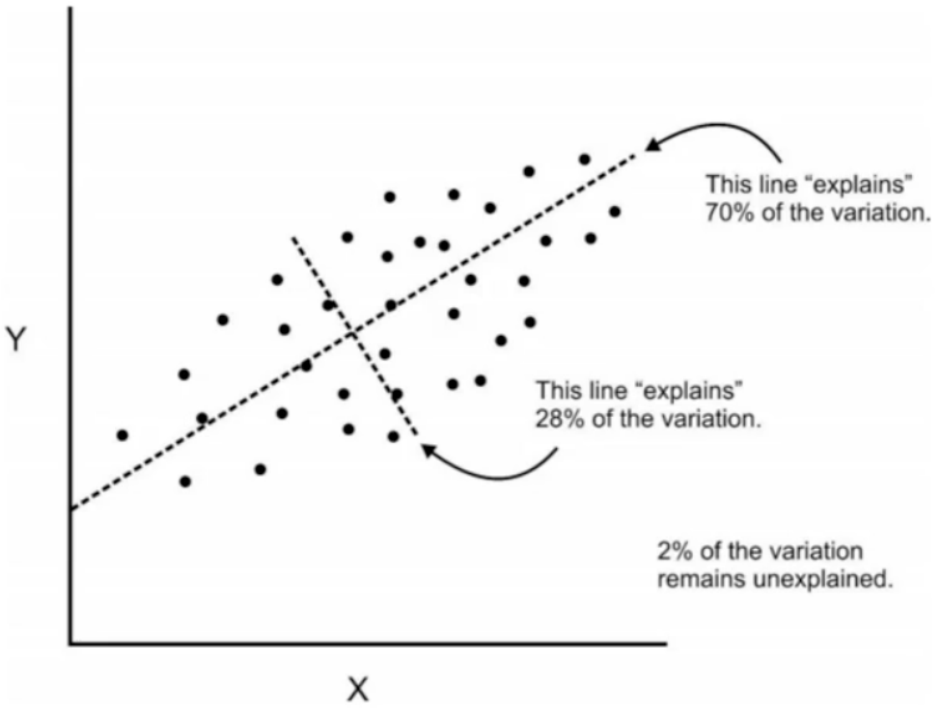
Principal Component Analysis

PCA summarises multiple fields of data into principal components, usually just 2 so that it is easier to visualise in a 2-dimensional plot. The 1st component will show the most variance of the entire dataset in the hyperplane, while the 2nd shows the 2nd shows the most variance at a right angle to the 1st. Because of the strong variance between data points, patterns tend to be teased out from a high dimension to even when there's just two dimensions. These 2 components can serve as new features for a supervised analysis.

In short, PCA finds the best possible characteristics, that summarises the classes of a feature. Two excellent sites elaborate more: setosa, quora. The most challenging part of PCA is interpreting the components.

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
df = pd.DataFrame(cancer['data'],columns=cancer['feature_names'])

# Before applying PCA, each feature should be centered (zero mean) and with unit␣
↪variance
scaled_data = StandardScaler().fit(df).transform(df)

pca = PCA(n_components = 2).fit(scaled_data)
# PCA(copy=True, n_components=2, whiten=False)

x_pca = pca.transform(scaled_data)
print(df.shape, x_pca.shape)

# RESULTS
(569, 30) (569, 2)
```

To see how much variance is preserved for each dataset.




```
percent = pca.explained_variance_ratio_
print(percent)
print(sum(percent))

# [0.9246348, 0.05238923] 1st component explained variance of 92%, 2nd explained 5%
# 0.986 total variance explained from 2 components is 97%
```

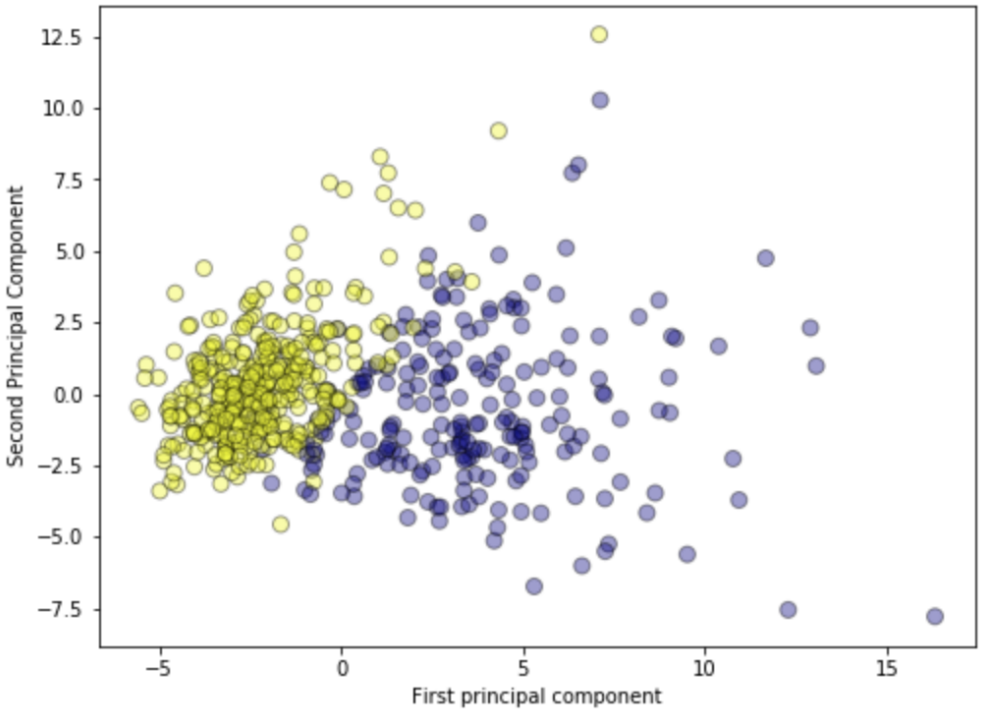
Plotting the PCA-transformed version of the breast cancer dataset. We can see that malignant and benign cells cluster between two groups and can apply a linear classifier to this two dimensional representation of the dataset.

```
plt.figure(figsize=(8,6))
plt.scatter(x_pca[:,0], x_pca[:,1], c=cancer['target'], cmap='plasma', alpha=0.4,␣
↪edgecolors='black', s=65);
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
```

Plotting the magnitude of each feature value for the first two principal components. This gives the best explanation for the components for each field.

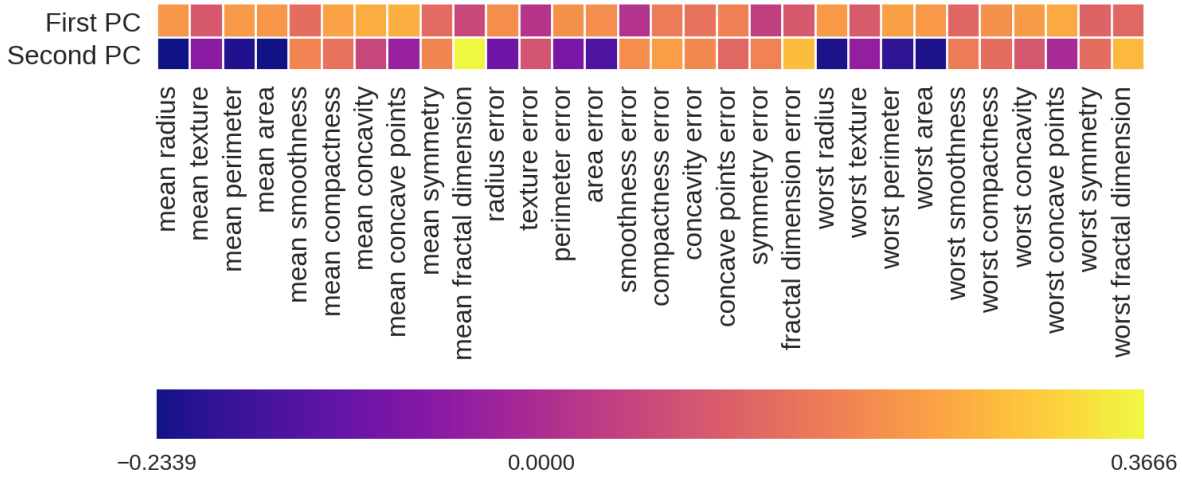
```
fig = plt.figure(figsize=(8, 4))
plt.imshow(pca.components_, interpolation = 'none', cmap = 'plasma')
feature_names = list(cancer.feature_names)

plt.gca().set_xticks(np.arange(-.5, len(feature_names)));
plt.gca().set_yticks(np.arange(0.5, 2));
plt.gca().set_xticklabels(feature_names, rotation=90, ha='left', fontsize=12);
plt.gca().set_yticklabels(['First PC', 'Second PC'], va='bottom', fontsize=12);
```

(continues on next page)



(continued from previous page)

```
plt.colorbar(orientation='horizontal', ticks=[pca.components_.min(), 0,
                                              pca.components_.max()], pad=0.65);
```



We can also plot the feature magnitudes in the scatterplot like in R into two separate axes, also known as a biplot. This shows the relationship of each feature's magnitude clearer in a 2D space.

```
# put feature values into dataframe
components = pd.DataFrame(pca.components_.T, index=df.columns, columns=['PCA1','PCA2
↪'])
```

(continues on next page)

(continued from previous page)

```
# plot size
plt.figure(figsize=(10,8))

# main scatterplot
plt.scatter(x_pca[:,0], x_pca[:,1], c=cancer['target'], cmap='plasma', alpha=0.4,␣
↪edgecolors='black', s=40);
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.ylim(15,-15);
plt.xlim(20,-20);

# individual feature values
ax2 = plt.twinx().twiny();
ax2.set_ylim(-0.5,0.5);
ax2.set_xlim(-0.5,0.5);

# reference lines
ax2.hlines(0,-0.5,0.5, linestyles='dotted', colors='grey')
ax2.vlines(0,-0.5,0.5, linestyles='dotted', colors='grey')

# offset for labels
offset = 1.07

# arrow & text
for a, i in enumerate(components.index):
    ax2.arrow(0, 0, components['PCA1'][a], -components['PCA2'][a], \
              alpha=0.5, facecolor='white', head_width=.01)
    ax2.annotate(i, (components['PCA1'][a]*offset, -components['PCA2'][a]*offset),␣
↪color='orange')
```

Lastly, we can specify the percentage explained variance, and let PCA decide on the number components.

```
from sklearn.decomposition import PCA
pca = PCA(0.99)
df_pca = pca.fit_transform(df)

# check no. of resulting features
df_pca.shape
```

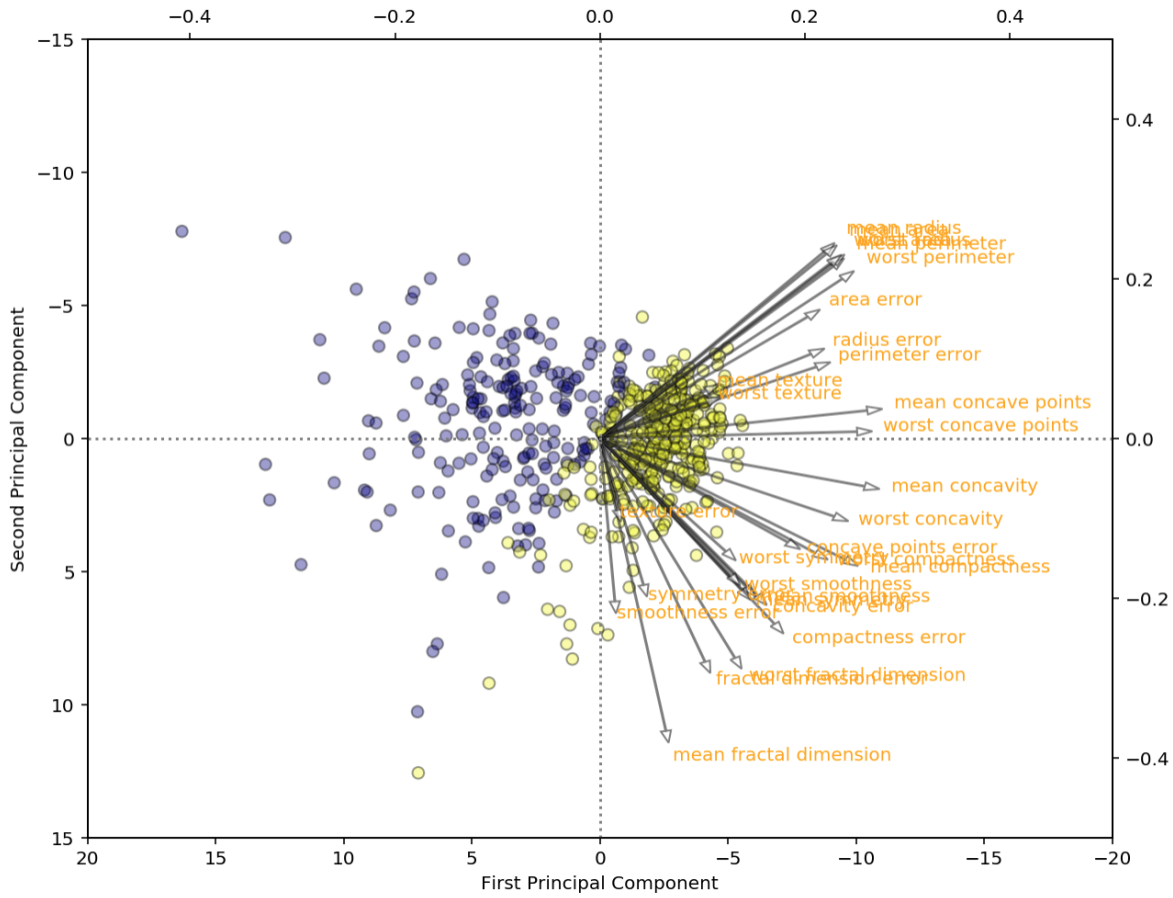
Multi-Dimensional Scaling

Multi-Dimensional Scaling (MDS) is a type of manifold learning algorithm that to visualize a high dimensional dataset and project it onto a lower dimensional space - in most cases, a two-dimensional page. PCA is weak in this aspect.

sklearn gives a good overview of various manifold techniques. https://scikit-learn.org/stable/modules/manifold.html

```
from adspy_shared_utilities import plot_labelled_scatter
from sklearn.preprocessing import StandardScaler
from sklearn.manifold import MDS

# each feature should be centered (zero mean) and with unit variance
X_fruits_normalized = StandardScaler().fit(X_fruits).transform(X_fruits)

mds = MDS(n_components = 2)
```

(continues on next page)


Second Principal Component
First Principal Component
mean perimeter
worst perimeter
area error
radius error
perimeter error
mean texture
worst texture
mean concave points
worst concave points
mean concavity
worst concavity
concave points error
mean compactness
worst smoothness
mean smoothness
concavity error
compactness error
fractal dimension error
mean fractal dimension

(continued from previous page)

```
X_fruits_mds = mds.fit_transform(X_fruits_normalized)

plot_labelled_scatter(X_fruits_mds, y_fruits, ['apple', 'mandarin', 'orange', 'lemon
↪'])
plt.xlabel('First MDS feature')
plt.ylabel('Second MDS feature')
plt.title('Fruit sample dataset MDS');
```

t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a powerful manifold learning algorithm for visualizing clusters. It finds a two-dimensional representation of your data, such that the distances between points in the 2D scatterplot match as closely as possible the distances between the same points in the original high dimensional dataset. In particular, t-SNE gives much more weight to preserving information about distances between points that are neighbors.

More information here.

```
from sklearn.manifold import TSNE

tsne = TSNE(random_state = 0)

X_tsne = tsne.fit_transform(X_fruits_normalized)

plot_labelled_scatter(X_tsne, y_fruits,
```

(continues on next page)

(continued from previous page)

```
    ['apple', 'mandarin', 'orange', 'lemon'])
plt.xlabel('First t-SNE feature')
plt.ylabel('Second t-SNE feature')
plt.title('Fruits dataset t-SNE');
```

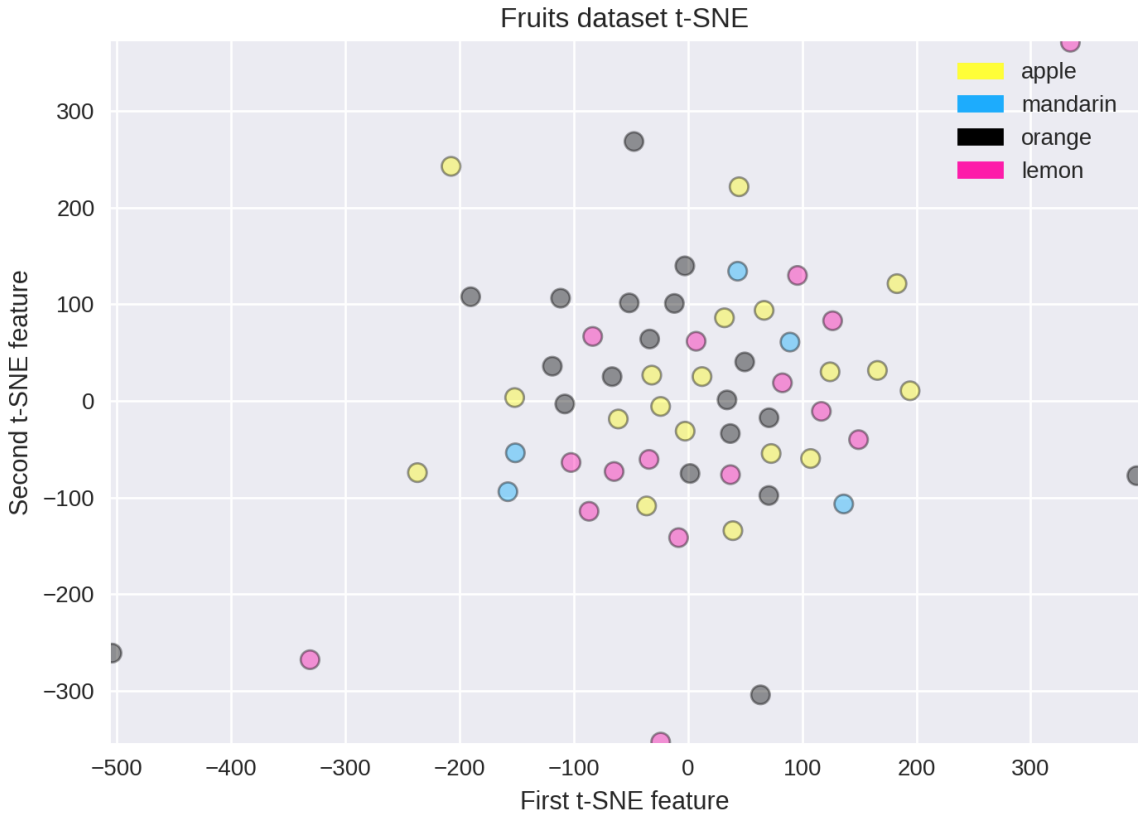


Fig. 2: You can see how some dimensionality reduction methods may be less successful on some datasets. Here, it doesn't work as well at finding structure in the small fruits dataset, compared to other methods like MDS.

LDA

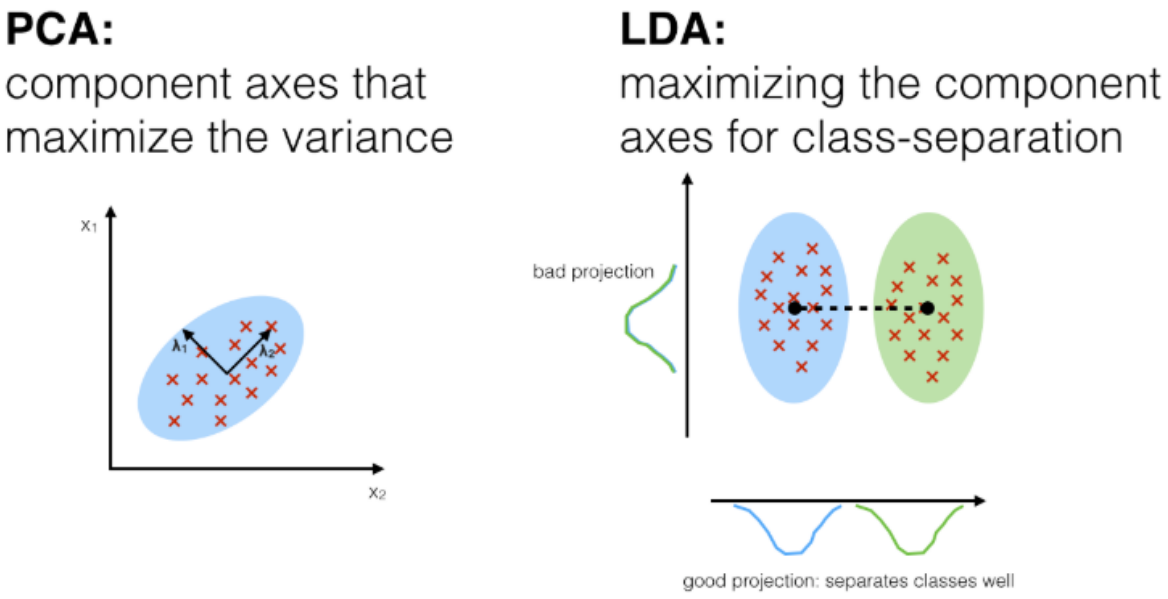
Latent Dirichlet Allocation is another dimension reduction method, but unlike PCA, it is a supervised method. It attempts to find a feature subspace or decision boundary that maximizes class separability. It then projects the data points to new dimensions in a way that the clusters are as separate from each other as possible and the individual elements within a cluster are as close to the centroid of the cluster as possible.

Differences of PCA & LDA, from:

- https://sebastianraschka.com/Articles/2014_python_lda.html
- https://stackabuse.com/implementing-lda-in-python-with-scikit-learn/

```
# from sklearn documentation
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.datasets import make_multilabel_classification
```

(continues on next page)



(continued from previous page)

```
# This produces a feature matrix of token counts, similar to what
# CountVectorizer would produce on text.
X, _ = make_multilabel_classification(random_state=0)
lda = LatentDirichletAllocation(n_components=5, random_state=0)
X_lda = lda.fit_transform(X, y)

# check the explained variance
percent = lda.explained_variance_ratio_
print(percent)
print(sum(percent))
```

Self-Organzing Maps

SOM is a special type of neural network that is trained using unsupervised learning to produce a two-dimensional map. Each row of data is assigned to its Best Matching Unit (BMU) neuron. Neighbourhood effect to create a topographic map

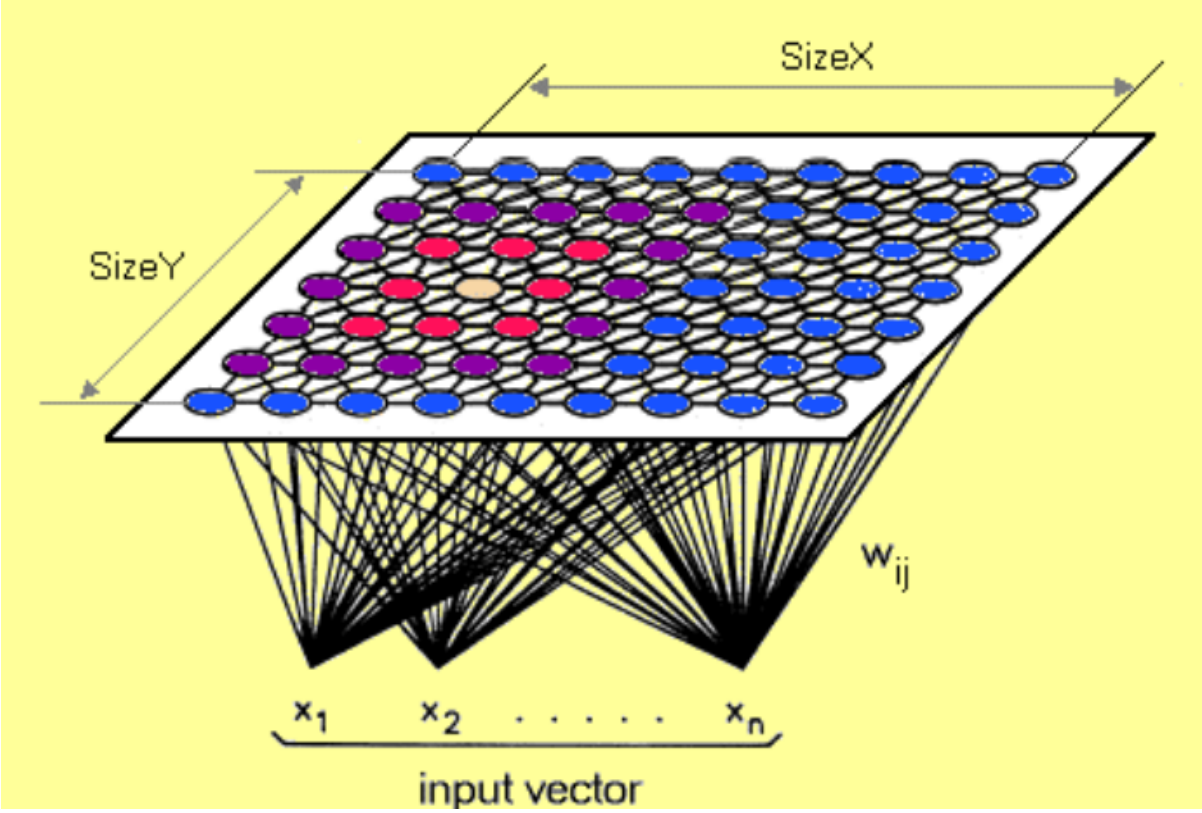
They differ from other artificial neural networks as:

1. they apply competitive learning as opposed to error-correction learning (such as backpropagation with gradient descent)
2. in the sense that they use a neighborhood function to preserve the topological properties of the input space.
3. Consist of only one visible output layer

Requires scaling or normalization of all features first.

https://github.com/JustGlowing/minisom

We first need to calculate the number of neurons and how many of them making up each side. The ratio of the side lengths of the map is approximately the ratio of the two largest eigenvalues of the training data's covariance matrix.




```
# total no. of neurons required
total_neurons = 5*sqrt(normal.shape[1])

# calculate eigen_values
normal_cov = np.cov(data_normal)
eigen_values = np.linalg.eigvals(normal_cov)

# 2 largest eigenvalues
result = sorted([i.real for i in eigen_values])[-2:]
ratio_2_largest_eigen = result[1]/result[0]

side = total_neurons/ratio_2_largest_eigen

# two sides
print(total_neurons)
print('1st side', side)
print('2nd side', ratio_2_largest_eigen)
```

Then we build the model.

```
# 1st side, 2nd side, # features
model = MiniSom(5, 4, 66, sigma=1.5, learning_rate=0.5,
                neighborhood_function='gaussian', random_seed=10)

# initialise weights to the map
model.pca_weights_init(data_normal)
# train the model
model.train_batch(df, 60000, verbose=True)
```

Plot out the map.

```
plt.figure(figsize=(6, 5))
plt.pcolor(som.distance_map().T, cmap='bone_r')
```

Quantization error is the distance between each vector and the BMU.

```
som.quantization_error(array)
```

10.2 Clustering

Find groups in data & assign every point in the dataset to one of the groups.

10.2.1 K-Means

Need to specify K number of clusters. It is also important to scale the features before applying K-means, unless the fields are not meant to be scaled, like distances. Categorical data is not appropriate as clustering calculated using euclidean distance (means). For long distances over an lat/long coordinates, they need to be projected to a flat surface.

One aspect of k means is that different random starting points for the cluster centers often result in very different clustering solutions. So typically, the k-means algorithm is run in scikit-learn with ten different random initializations and the solution occurring the most number of times is chosen.

Downsides

- Very sensitive to outliers. Have to remove before running the model
- Might need to reduce dimensions if very high no. of features or the distance separation might not be obvious

Methodology

1. Specify number of clusters (3)
2. 3 random data points are randomly selected as cluster centers
3. Each data point is assigned to the cluster center it is cloest to
4. Cluster centers are updated to the mean of the assigned points
5. Steps 3-4 are repeated, till cluster centers remain unchanged

Example 1

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from adspy_shared_utilities import plot_labelled_scatter
from sklearn.preprocessing import MinMaxScaler

fruits = pd.read_table('fruit_data_with_colors.txt')
X_fruits = fruits[['mass','width','height', 'color_score']].as_matrix()
y_fruits = fruits[['fruit_label']] - 1

X_fruits_normalized = MinMaxScaler().fit(X_fruits).transform(X_fruits)

kmeans = KMeans(n_clusters = 4, random_state = 0)
kmeans.fit(X_fruits)
```

(continues on next page)

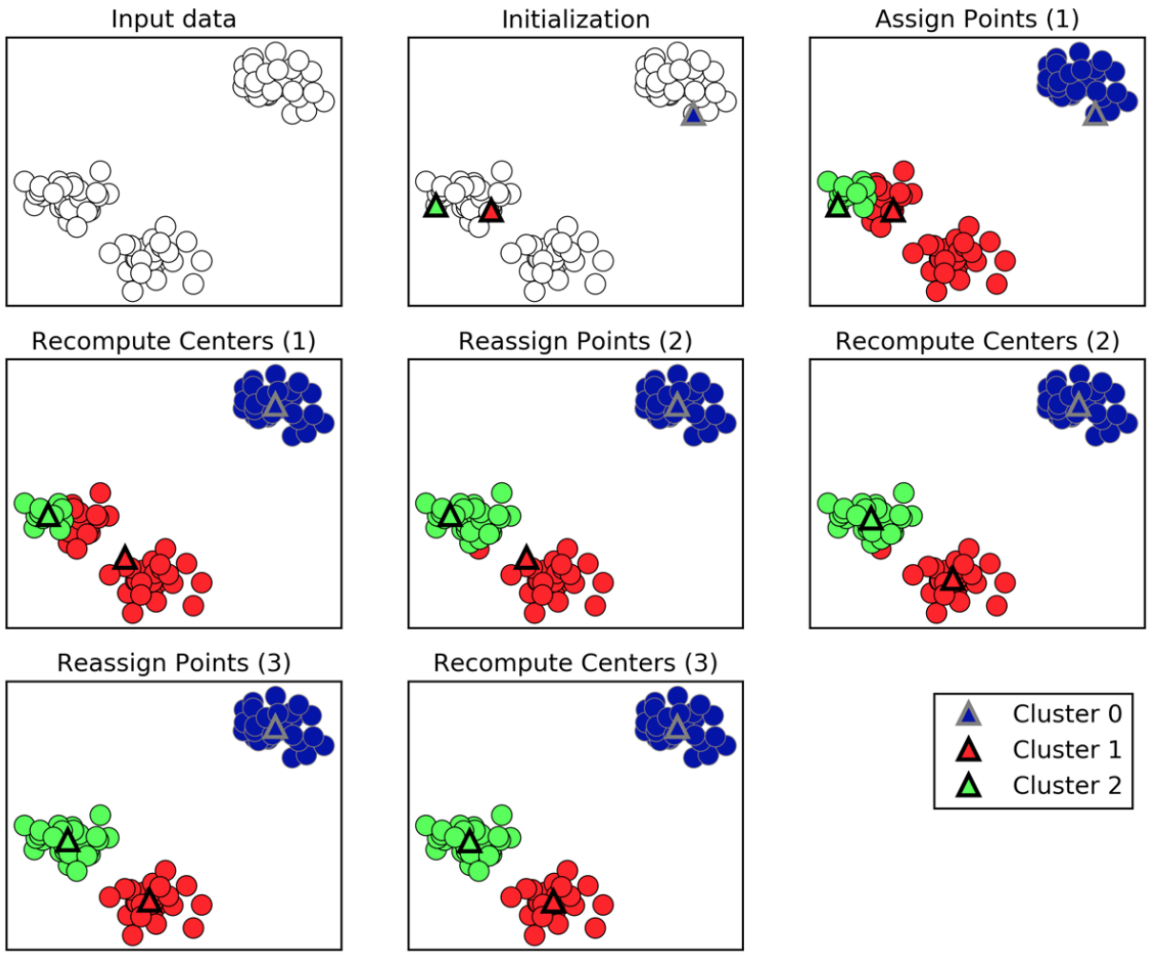


Fig. 3: Introduction to Machine Learning with Python

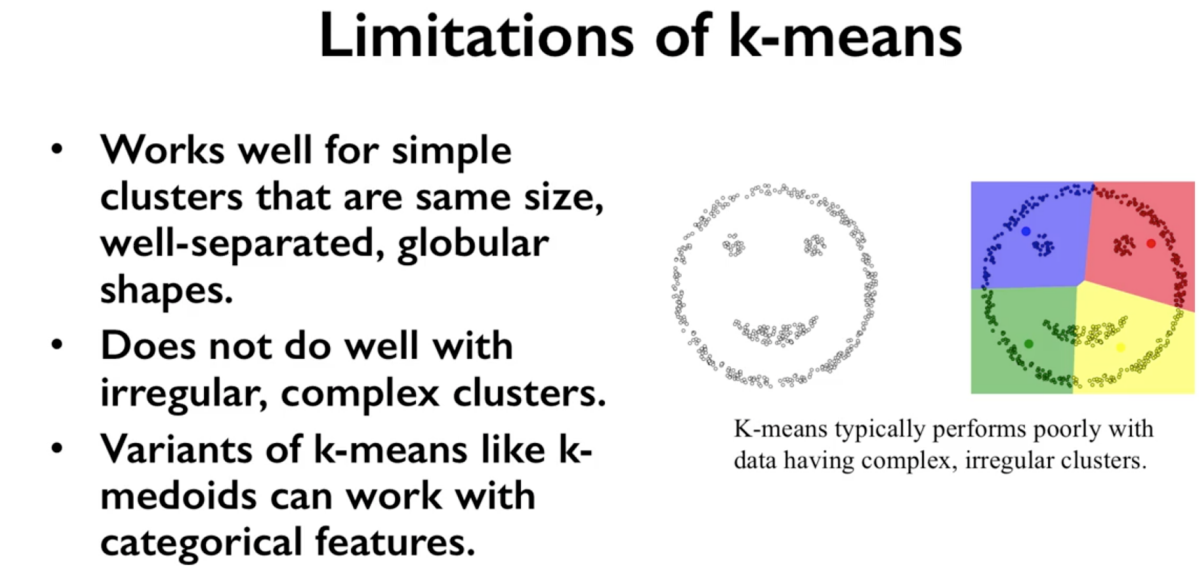


Fig. 4: University of Michigan: Coursera Data Science in Python

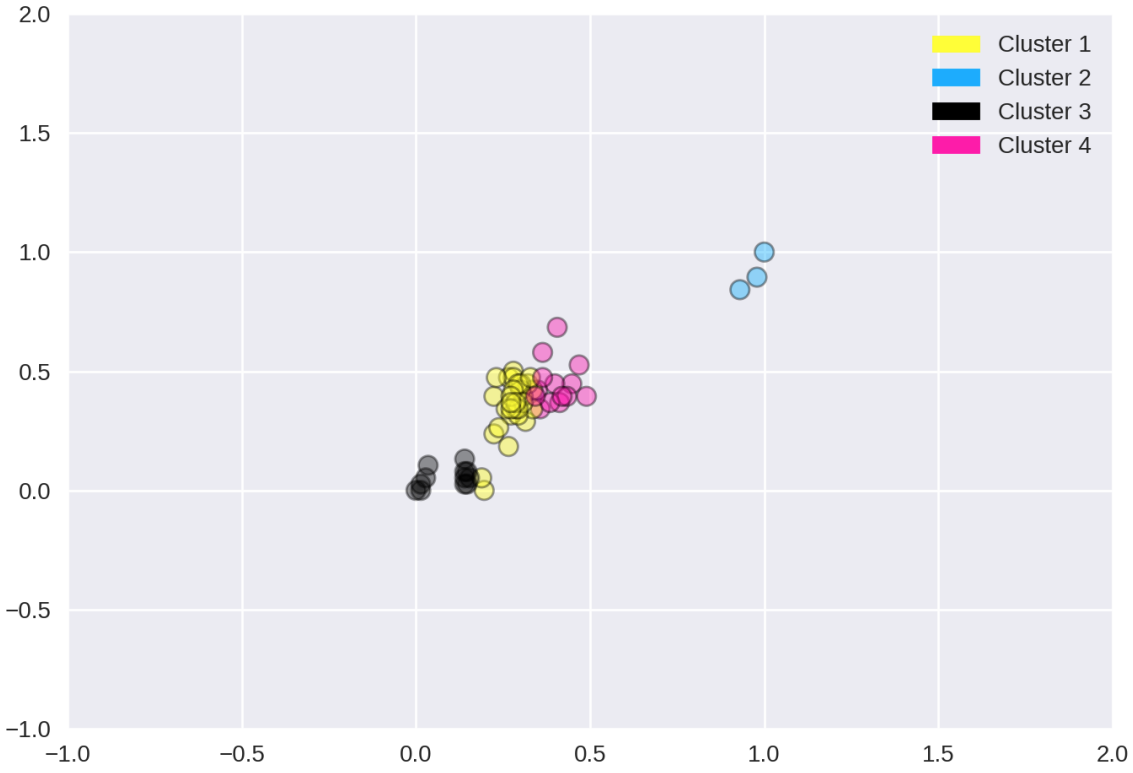
(continued from previous page)

```
plot_labelled_scatter(X_fruits_normalized, kmeans.labels_,
                      ['Cluster 1', 'Cluster 2', 'Cluster 3', 'Cluster 4'])
```

Example 2

```
#### IMPORT MODULES ####
import pandas as pd
from sklearn import preprocessing
from sklearn.cross_validation import train_test_split
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris



#### NORMALIZATION ####
# standardise the means to 0 and standard error to 1
for i in df.columns[:-2]: # df.columns[:-1] = dataframe for all features, minus target
    df[i] = preprocessing.scale(df[i].astype('float64'))

df.describe()



#### TRAIN-TEST SPLIT ####
train_feature, test_feature = train_test_split(feature, random_state=123, test_size=0.
↪2)

print train_feature.shape
print test_feature.shape
(120, 4)
(30, 4)
```

(continues on next page)



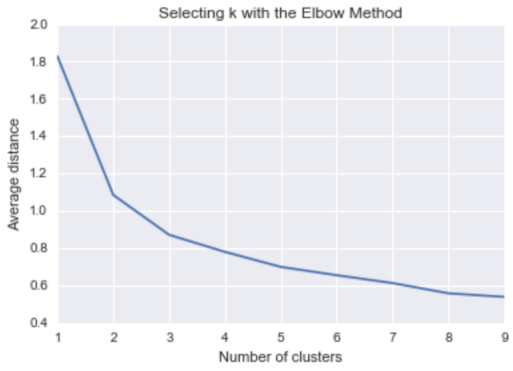
(continued from previous page)

```
#### A LOOK AT THE MODEL ####
KMeans(n_clusters=2)
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=2, n_init=10,
    n_jobs=1, precompute_distances='auto', random_state=None, tol=0.0001,
    verbose=0)



#### ELBOW CHART TO DETERMINE OPTIMUM K ####
from scipy.spatial.distance import cdist
import numpy as np
clusters=range(1,10)
# to store average distance values for each cluster from 1-9
meandist=[]

# k-means cluster analysis for 9 clusters
for k in clusters:
    # prepare the model
    model=KMeans(n_clusters=k)
    # fit the model
    model.fit(train_feature)
    # test the model
    clusassign=model.predict(train_feature)
    # gives average distance values for each cluster solution
        # cdist calculates distance of each two points from centriod
```

(continues on next page)

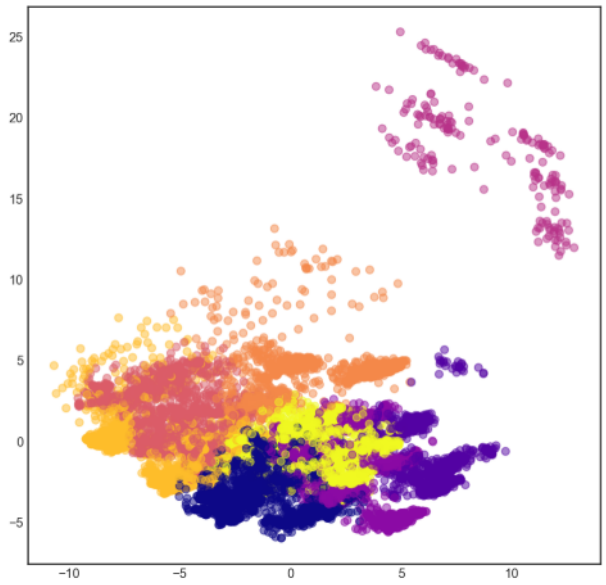
(continued from previous page)

```
        # get the min distance (where point is placed in clsuter)
        # get average distance by summing & dividing by total number of samples
    meandist.append(sum(np.min(cdist(train_feature, model.cluster_centers_, 'euclidean
↪'), axis=1))
    / train_feature.shape[0])


import matplotlib.pylab as plt
import seaborn as sns
%matplotlib inline
"""Plot average distance from observations from the cluster centroid
to use the Elbow Method to identify number of clusters to choose"""

plt.plot(clusters, meandist)
plt.xlabel('Number of clusters')
plt.ylabel('Average distance')
plt.title('Selecting k with the Elbow Method')

# look a bend in the elbow that kind of shows where
# the average distance value might be leveling off such that adding more clusters
# doesn't decrease the average distance as much
```



We can visualise the clusters by reducing the dimensions into 2 using PCA. They are separate by theissen polygons, though at a multi-dimensional space.

```
pca = PCA(n_components = 2).fit(df).transform(df)
labels = kmeans.labels_

plt.figure(figsize=(8,8))
plt.scatter(pd.DataFrame(pca)[0],pd.DataFrame(pca)[1], c=labels, cmap='plasma',␣
↪alpha=0.5);
```



Sometimes we need to find the cluster centres so that we can get an absolute distance measure of centroids to new data. Each feature will have a defined centre for each cluster.

```
# get cluster centres
centroids = model.cluster_centers_
# for each row, define cluster centre
centroid_labels = [centroids[i] for i in model.labels_]
```

If we have labels or y, and want to determine which y belongs to which cluster for an evaluation score, we can use a groupby to find the most number of labels that fall in a cluster and manually label them as such.

df = concat.groupby(['label','cluster'])['cluster'].count() df

10.2.2 Gaussian Mixture Model

GMM is, in essence a density estimation model but can function like clustering. It has a probabilistic model under the hood so it returns a matrix of probabilities belonging to each cluster for each data point. More: https://jakevdp.github.io/PythonDataScienceHandbook/05.12-gaussian-mixtures.html

We can input the *covariance_type* argument such that it can choose between *diag* (the default, ellipse constrained to the axes), *spherical* (like k-means), or *full* (ellipse without a specific orientation).

```
from sklearn.mixture import GaussianMixture

# gmm accepts input as array, so have to convert dataframe to numpy
input_gmm = normal.values

gmm = GaussianMixture(n_components=4, covariance_type='full', random_state=42)
gmm.fit(input_gmm)
result = gmm.predict(test_set)
```

BIC or *AIC* are used to determine the optimal number of clusters, the former usually recommends a simpler model. Note that number of clusters or components measures how well GMM works as a density estimator, not as a clustering algorithm.

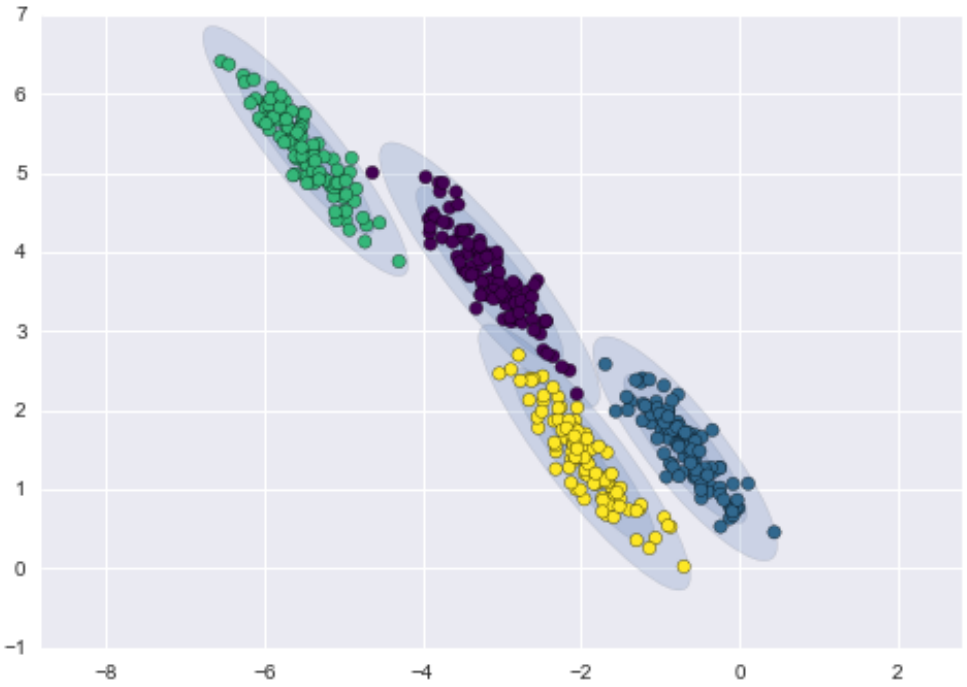


Fig. 5: from Python Data Science Handbook by Jake VanderPlas

```
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

input_gmm = normal.values

bic_list = []
aic_list = []
ranges = range(1,30)

for i in ranges:
    gmm = GaussianMixture(n_components=i).fit(input_gmm)
    # BIC
    bic = gmm.bic(input_gmm)
    bic_list.append(bic)
    # AIC
    aic = gmm.aic(input_gmm)
    aic_list.append(aic)

plt.figure(figsize=(10, 5))
plt.plot(ranges, bic_list, label='BIC');
plt.plot(ranges, aic_list, label='AIC');
plt.legend(loc='best');
```

10.2.3 Agglomerative Clustering

Agglomerative Clustering is a method of clustering technique used to build clusters from bottom up.

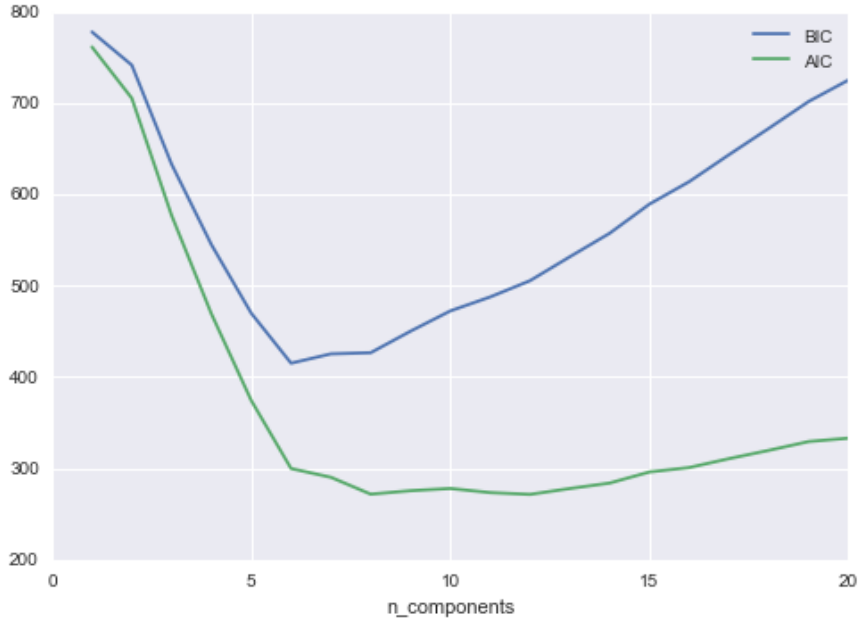


Fig. 6: from Python Data Science Handbook by Jake VanderPlas

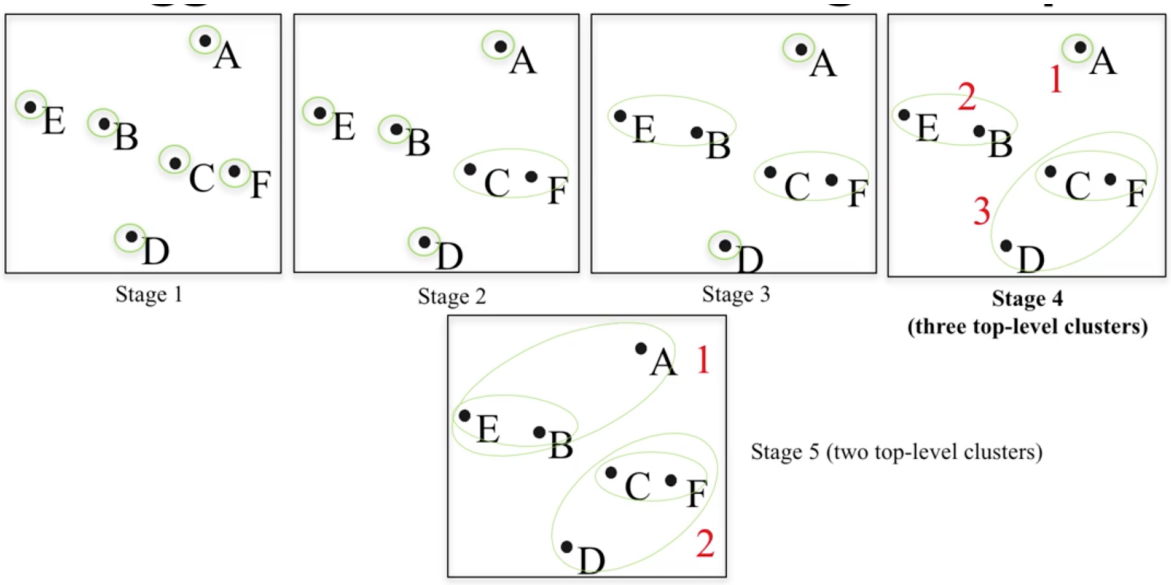


Fig. 7: University of Michigan: Coursera Data Science in Python

Methods of linking clusters together.

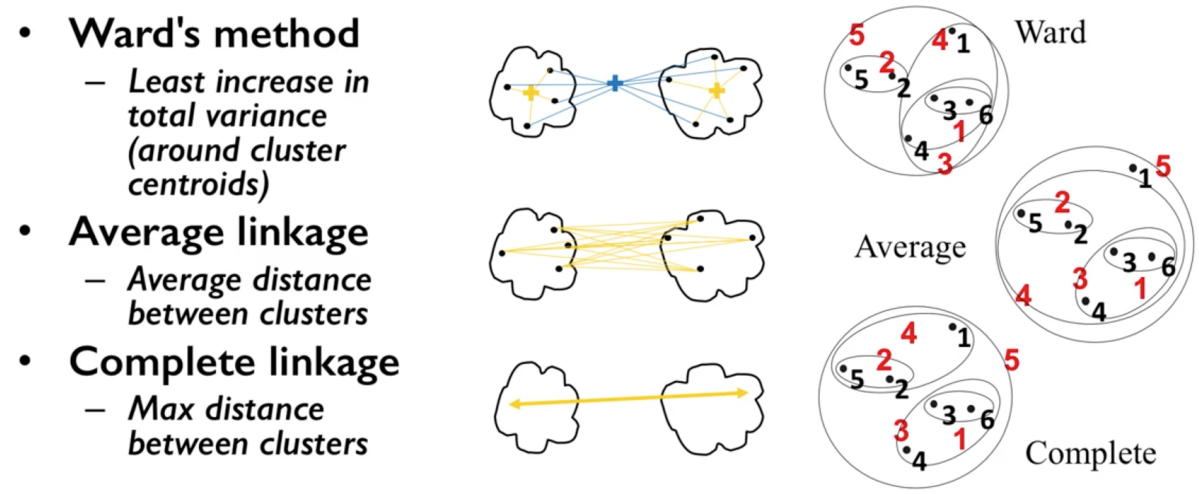
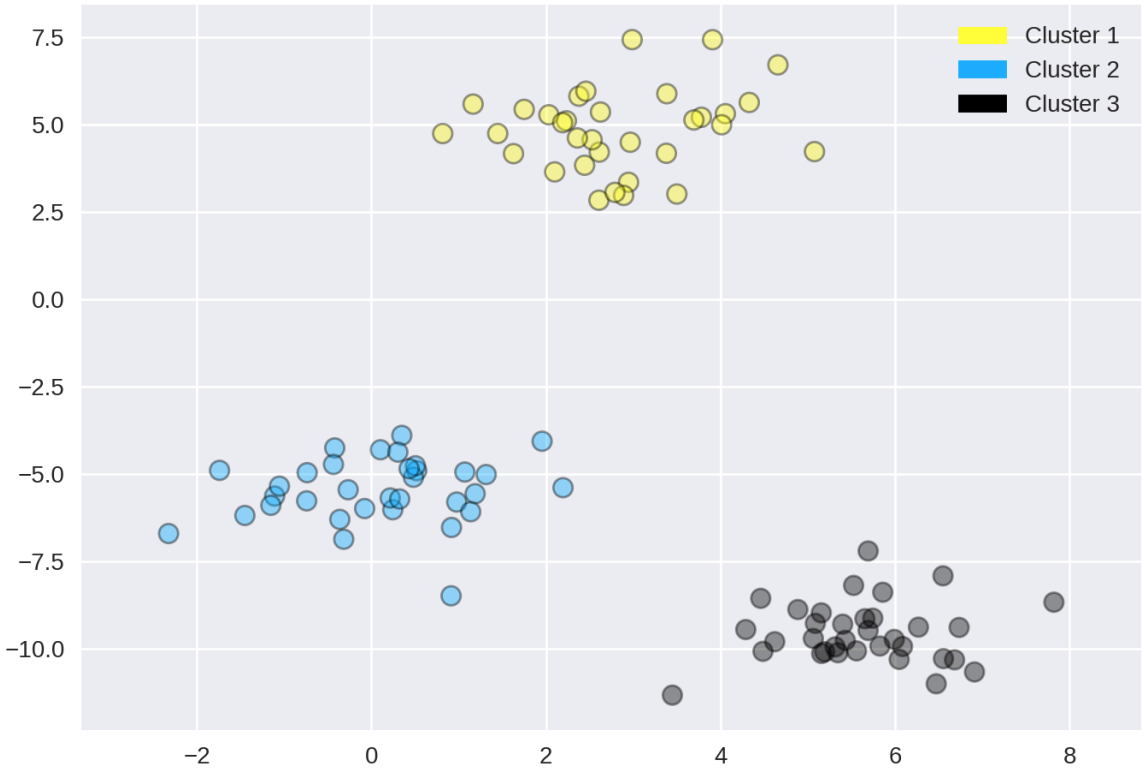


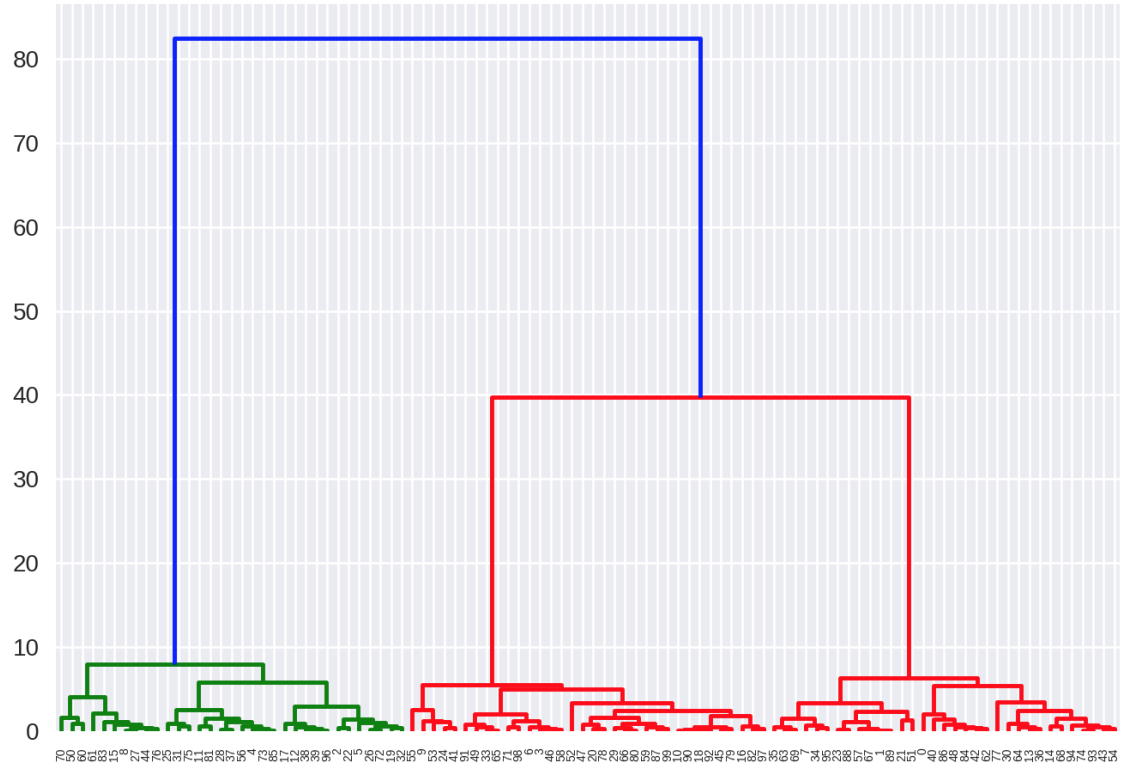
Fig. 8: University of Michigan: Coursera Data Science in Python

```
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
from adspy_shared_utilities import plot_labelled_scatter

X, y = make_blobs(random_state = 10)

cls = AgglomerativeClustering(n_clusters = 3)
cls_assignment = cls.fit_predict(X)

plot_labelled_scatter(X, cls_assignment,
        ['Cluster 1', 'Cluster 2', 'Cluster 3'])
```

One of the benfits of this clustering is that a hierarchy can be built.

```
X, y = make_blobs(random_state = 10, n_samples = 10)
plot_labelled_scatter(X, y,
        ['Cluster 1', 'Cluster 2', 'Cluster 3'])
print(X)

[[  5.69192445  -9.47641249]
 [  1.70789903   6.00435173]
 [  0.23621041  -3.11909976]
 [  2.90159483   5.42121526]
 [  5.85943906  -8.38192364]
 [  6.04774884 -10.30504657]
 [ -2.00758803  -7.24743939]
 [  1.45467725  -6.58387198]
 [  1.53636249   5.11121453]
 [  5.4307043   -9.75956122]]

 # BUILD DENDROGRAM
 from scipy.cluster.hierarchy import ward, dendrogram
 plt.figure(figsize=(10,5))
 dendrogram(ward(X))
 plt.show()
```


Cluster 1
Cluster 2
Cluster 3
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−2
0
2
4
6
8

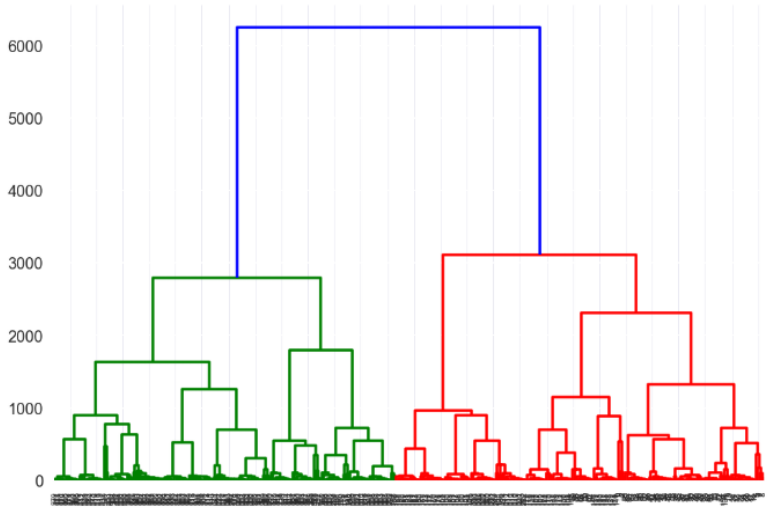

80
70
60
50
40
30
20
10
0

More in this link: https://joernhees.de/blog/2015/08/26/scipy-hierarchical-clustering-and-dendrogram-tutorial/

sklearn agglomerative clustering is very slow, and an alternative `fastcluster` library performs much faster as it is a C++ library with a python interface.

```
import fastcluster
from scipy.cluster.hierarchy import fcluster
from scipy.cluster.hierarchy import dendrogram, ward
from scipy.spatial.distance import pdist

Z = fastcluster.linkage_vector(df, method='ward', metric='euclidean')

# get dendrogram details into dataframe
Z_df = pd.DataFrame(data=Z, columns=['clusterOne','clusterTwo','distance',
↪'newClusterSize'])


# plot dendrogram
plt.figure(figsize=(10, 5))
dendrogram(ward(X))
plt.show();
```

	clusterOne	clusterTwo	distance	newClusterSize
0	231.0	232.0	2.970313	2.0
1	123.0	124.0	3.115567	2.0
2	203.0	204.0	3.361236	2.0
3	177.0	178.0	3.421791	2.0
4	267.0	268.0	3.505808	2.0



Then we select the distance threshold to cut the dendrogram to obtain the selected clustering level. The output is the cluster labelled for each row of data. As expected from the dendrogram, a cut at 2000 gives us 5 clusters.

```
distance_threshold = 2000
clusters = fcluster(Z, distance_threshold, criterion='distance')
chosen_clusters = pd.DataFrame(data=clusters, columns=['cluster'])

chosen_clusters['cluster'].unique()
# array([4, 5, 2, 3, 1], dtype=int64)
```

Evaluating the best number of clusters can be done through the elbow plot & BIC.

10.2.4 DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN). Need to scale/normalise data. DBSCAN works by identifying crowded regions referred to as dense regions.

Key parameters are `eps` and `min_samples`. If there are at least min_samples many data points within a distance of eps to a given data point, that point will be classified as a core sample. Core samples that are closer to each other than the distance eps are put into the same cluster by DBSCAN.

There is recently a new method called HDBSCAN (H = Hierarchical). https://hdbscan.readthedocs.io/en/latest/index.html

Methodology

1. Pick an arbitrary point to start
2. Find all points with distance *eps* or less from that point
3. If points are more than *min_samples* within distance of *esp*, point is labelled as a core sample, and assigned a new cluster label
4. Then all neighbours within *eps* of the point are visited
5. If they are core samples their neighbours are visited in turn and so on
6. The cluster thus grows till there are no more core samples within distance *eps* of the cluster
7. Then, another point that has not been visited is picked, and step 1-6 is repeated
8. 3 kinds of points are generated in the end, core points, boundary points, and noise
9. Boundary points are core clusters but not within distance of *esp*

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state = 9, n_samples = 25)

dbscan = DBSCAN(eps = 2, min_samples = 2)

cls = dbscan.fit_predict(X)
print("Cluster membership values:\n{}".format(cls))
Cluster membership values:
    [ 0  1  0  2  0  0  0  2  2 -1  1  2  0  0 -1  0  0  1 -1  1  1  2  2  2  1]
    # -1 indicates noise or outliers

plot_labelled_scatter(X, cls + 1,
        ['Noise', 'Cluster 0', 'Cluster 1', 'Cluster 2'])
```

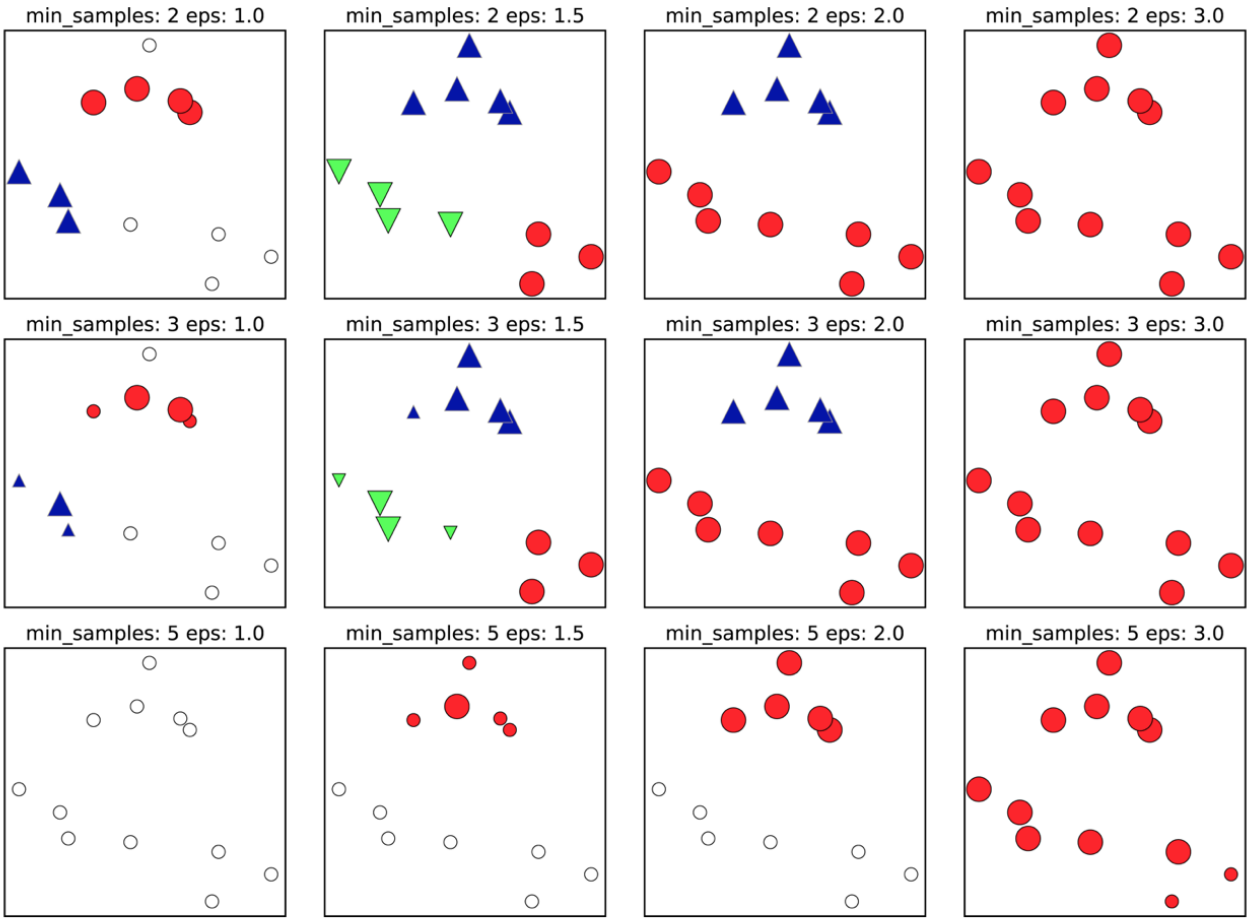


Figure 3-37. Cluster assignments found by DBSCAN with varying settings for the min_samples and eps parameters

Fig. 9: Introduction to Machine Learning with Python

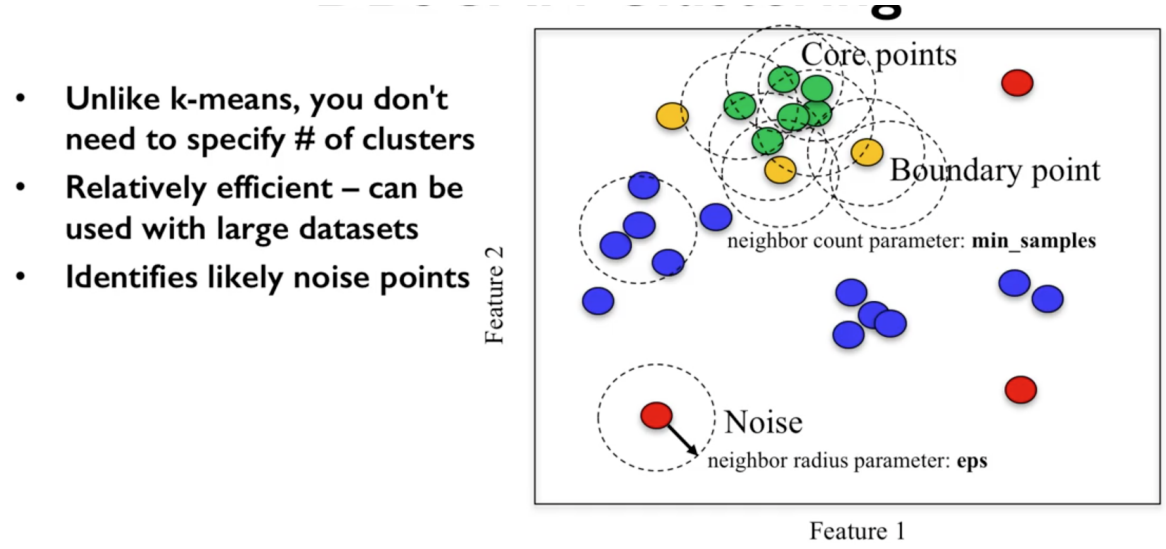
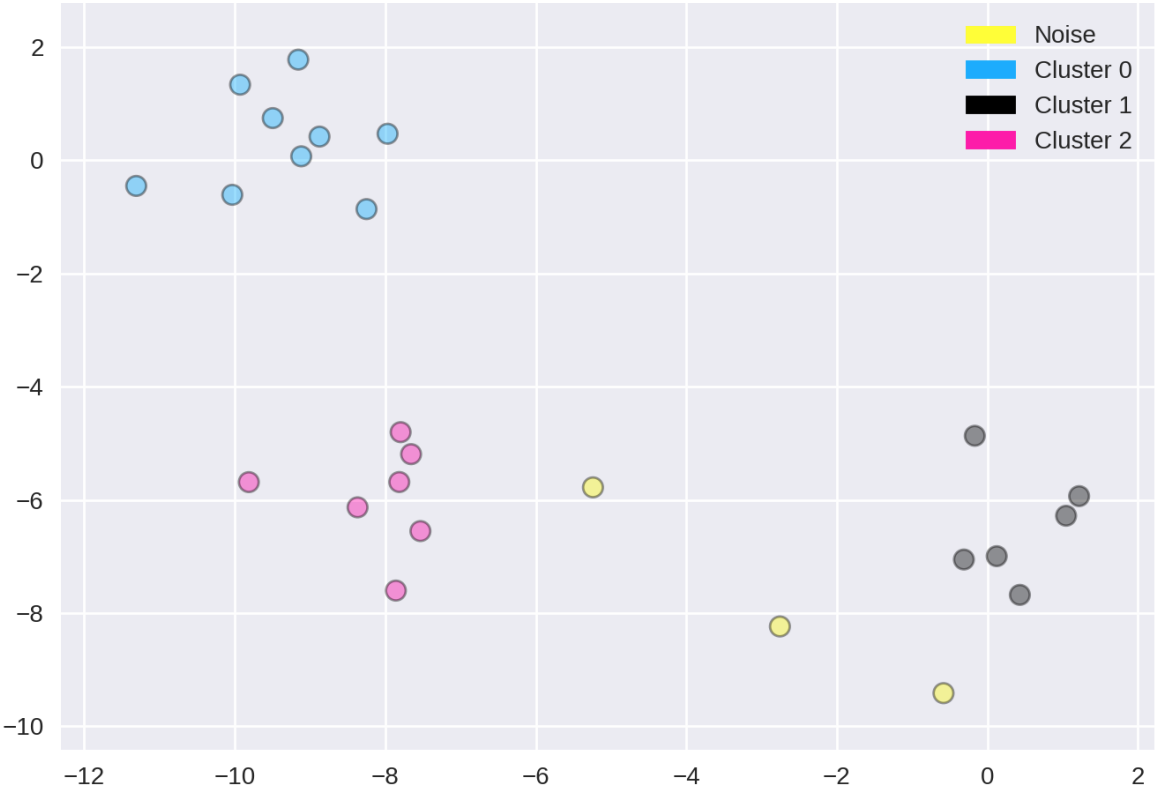


Fig. 10: University of Michigan: Coursera Data Science in Python



10.3 One-Class Classification

These requires the training of a normal state(s), allows outliers to be detected when they lie outside trained state.

10.3.1 One Class SVM

One-class SVM is an unsupervised algorithm that learns a decision function for outlier detection: classifying new data as similar or different to the training set.

Besides the kernel, two other parameters are impt: The nu parameter should be the proportion of outliers you expect to observe (in our case around 2%), the gamma parameter determines the smoothing of the contour lines.

```
from sklearn.svm import OneClassSVM

train, test = train_test_split(data, test_size=.2)
train_normal = train[train['y']==0]
train_outliers = train[train['y']==1]
outlier_prop = len(train_outliers) / len(train_normal)

model = OneClassSVM(kernel='rbf', nu=outlier_prop, gamma=0.000001)
svm.fit(train_normal[['x1','x4','x5']])
```

10.3.2 Isolation Forest

```
from sklearn.ensemble import IsolationForest

clf = IsolationForest(behaviour='new', max_samples=100,
                      random_state=rng, contamination='auto')

clf.fit(X_train)
y_pred_test = clf.predict(X_test)

# -1 are outliers
y_pred_test
# array([ 1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1,  1,  1])

# calculate the no. of anomalies
pd.DataFrame(save)[0].value_counts()
# -1    23330
# 1       687
# Name: 0, dtype: int64
```

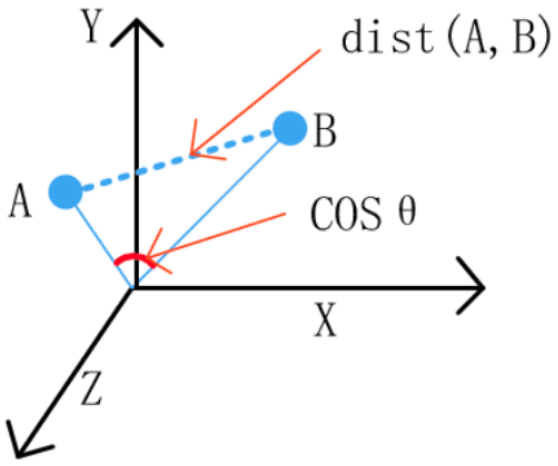
We can also get the average anomaly scores. The lower, the more abnormal. Negative scores represent outliers, positive scores represent inliers.

```
clf.decision_function(X_test)
array([ 0.14528263,  0.14528263, -0.08450298,  0.14528263,  0.14528263,
        0.14528263,  0.14528263,  0.14528263,  0.14528263, -0.14279962,
        0.14528263,  0.14528263, -0.05483886, -0.10086102,  0.14528263,
        0.14528263])
```

10.4 Distance Metrics

10.4.1 Euclidean Distance & Cosine Similarity

Euclidean distance is the straight line distance between points, while cosine distance is the cosine of the angle between these two points.




```
from scipy.spatial.distance import euclidean

euclidean([1,2],[1,3])
# 1
```

```
from scipy.spatial.distance import cosine

cosine([1,2],[1,3])
# 0.010050506338833642
```

10.4.2 Mahalanobis Distance

Mahalonobis distance is the distance between a point and a distribution, not between two distinct points. Therefore, it is effectively a multivariate equivalent of the Euclidean distance.

https://www.machinelearningplus.com/statistics/mahalanobis-distance/

- `x`: is the vector of the observation (row in a dataset),
- `m`: is the vector of mean values of independent variables (mean of each column),
- `C^(-1)`: is the inverse covariance matrix of independent variables.

Multiplying by the inverse covariance (correlation) matrix essentially means dividing the input with the matrix. This is so that if features in your dataset are strongly correlated, the covariance will be high. Dividing by a large covariance will effectively reduce the distance.

While powerful, its use of correlation can be detrimantal when there is multicollinearity (strong correlations among features).

$$\sqrt{(u-v)V^{-1}(u-v)^T}$$

```
import pandas as pd
import numpy as np
from scipy.spatial.distance import mahalanobis

def mahalanobisD(normal_df, y_df):
    # calculate inverse covariance from normal state
    x_cov = normal_df.cov()
    inv_cov = np.linalg.pinv(x_cov)

    # get mean of normal state df
    x_mean = normal_df.mean()

    # calculate mahalanobis distance from each row of y_df
    distanceMD = []
    for i in range(len(y_df)):
        MD = mahalanobis(x_mean, y_df.iloc[i], inv_cov)
        distanceMD.append(MD)

    return distanceMD
```

10.4.3 Jaccard's Distance

10.4.4 Dynamic Time Warping

If two time series are identical, but one is shifted slightly along the time axis, then Euclidean distance may consider them to be very different from each other. DTW was introduced to overcome this limitation and give intuitive distance measurements between time series by ignoring both global and local shifts in the time dimension.

DTW is a technique that finds the optimal alignment between two time series, if one time series may be "warped" non-linearly by stretching or shrinking it along its time axis. Dynamic time warping is often used in speech recognition to determine if two waveforms represent the same spoken phrase. In a speech waveform, the duration of each spoken sound and the interval between sounds are permitted to vary, but the overall speech waveforms must be similar.

From the creators of FastDTW, it produces an accurate minimum-distance warp path between two time series than is nearly optimal (standard DTW is optimal, but has a quadratic time and space complexity).

Output: Identical = 0, Difference > 0

```
import numpy as np
from scipy.spatial.distance import euclidean
from fastdtw import fastdtw

x = np.array([[1,1], [2,2], [3,3], [4,4], [5,5]])
y = np.array([[2,2], [3,3], [4,4]])
distance, path = fastdtw(x, y, dist=euclidean)
print(distance)

# 2.8284271247461903
```

Stan Salvador & Philip ChanFast. DTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. Florida Institude of Technology. https://cs.fit.edu/~pkc/papers/tdm04.pdf

CHAPTER 11

Active Learning

11.1 Introduction

Getting labeled data is a huge and often prohibitive cost for a lot of machine learning projects. Active Learning is a methodology that can sometimes greatly reduce the amount of labeled data required to train a model with higher accuracy if it is allowed to choose which data to label. It does this by prioritizing the labeling work for the experts (oracles).

Active Learning prioritizes which data the model is most confused about and requests labels for just those. This helps the model learn faster, and lets the experts skip labeling data that wouldn't be very helpful to the model.

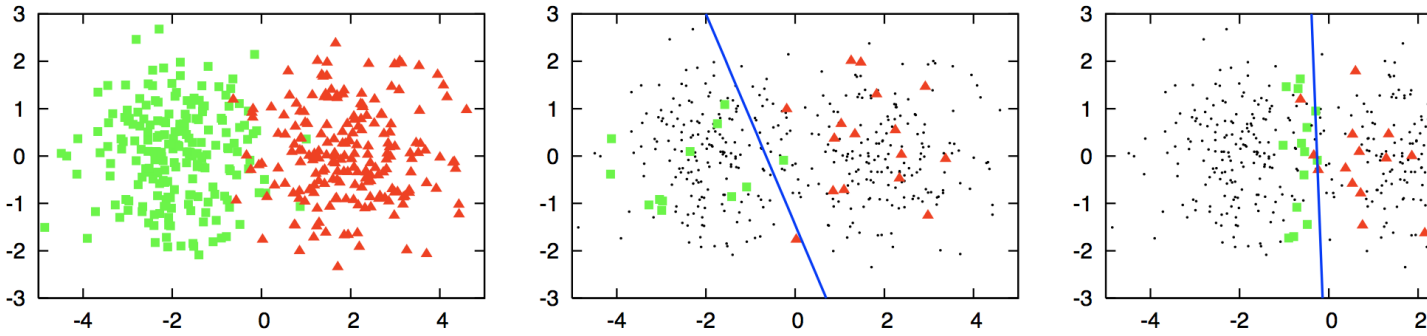


Fig. 1: Training samples near the decision boundary allows a more accurate hyperplane being drawn.

11.2 Sampling Methods

1. **Membership Query Synthesis**: a synethsized sample is sent to an oracle for labeling.
2. **Stream-Based Selective Sampling**: each sample is presented one at a time to a predictive model to be decided whether or not to be labeled or not. There are no assumptions on data distribution, and therefore it is adaptive

to change.

3. **Pool-Based Sampling**: This is similar to stream-based, except that it starts a large pool of unlabelled data.

The main difference between stream-based and pool-based active learning is that the former scans through the data sequentially and makes query decisions individually, whereas the latter evaluates and ranks the entire collection before selecting the best query.

11.3 Query Strategies

1. **Uncertainiy Sampling**: Learner will choose instances which it is least certain how to label. There are 3 methods in this sampling, i.e., Least Confidence, Margin Sampling, and Entropy Sampling, with the latter being the best among the 3 due to its consideration of utilizing all the possible label probabilities for the selection process.
2. **Query by Committee**: Using an ensemble of models to vote on which candidates to label.

11.4 Stop Criteria

We can use a performance metric, e.g. accuracy to determine when to stop further querying. Ideally it should be when any further labelling and retraining of the model does not improve the performance metric significantly, i.e., the performance has reached plateau. This means that the slope of the graph (y2-y1/x2-x1) per time-step has neared 0.

Of course, plateauing is determined on a good query strategy together with an appropriate model.

11.5 Resources

- Active Learning Literature Survey. http://burrsettles.com/pub/settles.activelearning.pdf
- Class Imbalance & Active Learning. https://pdfs.semanticscholar.org/7437/aae9bf347ab4ba4057f28df5f2eaf64d8fdc.pdf

CHAPTER 12

Deep Learning

Deep Learning falls under the broad class of Articial Intelligence > Machine Learning. It is a Machine Learning technique that uses multiple internal layers (**hidden layers**) of non-linear processing units (**neurons**) to conduct supervised or unsupervised learning from data.

12.1 Introduction

12.1.1 GPU

Tensorflow is able to run faster and more effeciently using Nivida's GPU `pip install tensorflow-gpu`.

12.1.2 Preprocessing

Keras accepts numpy input, so we have to convert. Also, for multi-class classification, we need to convert them into binary values; i.e., using one-hot encoding. For the latter, we can in-place use `sparse_categorical_crossentropy` for the loss function which will can process the multi-class label without converting to one-hot encoding.

```
# convert to numpy arrays
X = np.array(X)
# OR
X = X.values

# one-hot encoding for multi-class y labels
Y = pd.get_dummies(y)
```

It is important to scale or normalise the dataset before putting in the neural network.

```
from sklearn.preprocessing import StandardScaler

X_train, X_test, y_train, y_test = train_test_split(X, y,
```

(continues on next page)

(continued from previous page)

```
                                                    random_state = 0)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Model architecture can also be displayed in a graph. Or we can print as a summary

```
from IPython.display import SVG
from tensorflow.python.keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
```

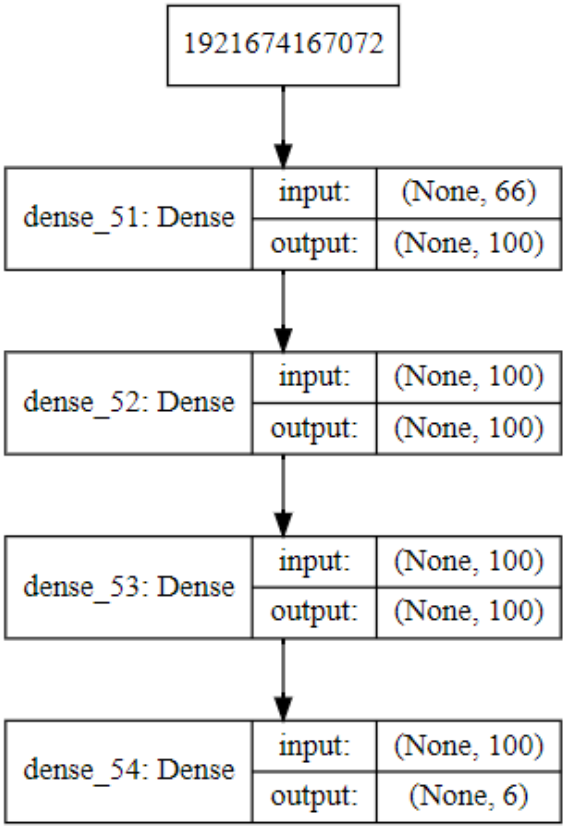


Fig. 1: model architecture printout

```
model.summary()
```

12.1.3 Evaluation

The model compiled has a history method (`model.history.history`) that gives the accuracy and loss for both train & test sets for each time step. We can plot it out for a better visualization. Alternatively we can also use TensorBoard, which is installed together with TensorFlow package. It will also draw the model architecture.

```
def plot_validate(model, loss_acc):
    '''Plot model accuracy or loss for both train and test validation per epoch
```

(continues on next page)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_63 (Dense)             (None, 100)               6700
_________________________________________________________________
dense_64 (Dense)             (None, 100)               10100
_________________________________________________________________
dense_65 (Dense)             (None, 100)               10100
_________________________________________________________________
dense_66 (Dense)             (None, 6)                 606
=================================================================
Total params: 27,506
Trainable params: 27,506
Non-trainable params: 0
_________________________________________________________________
```

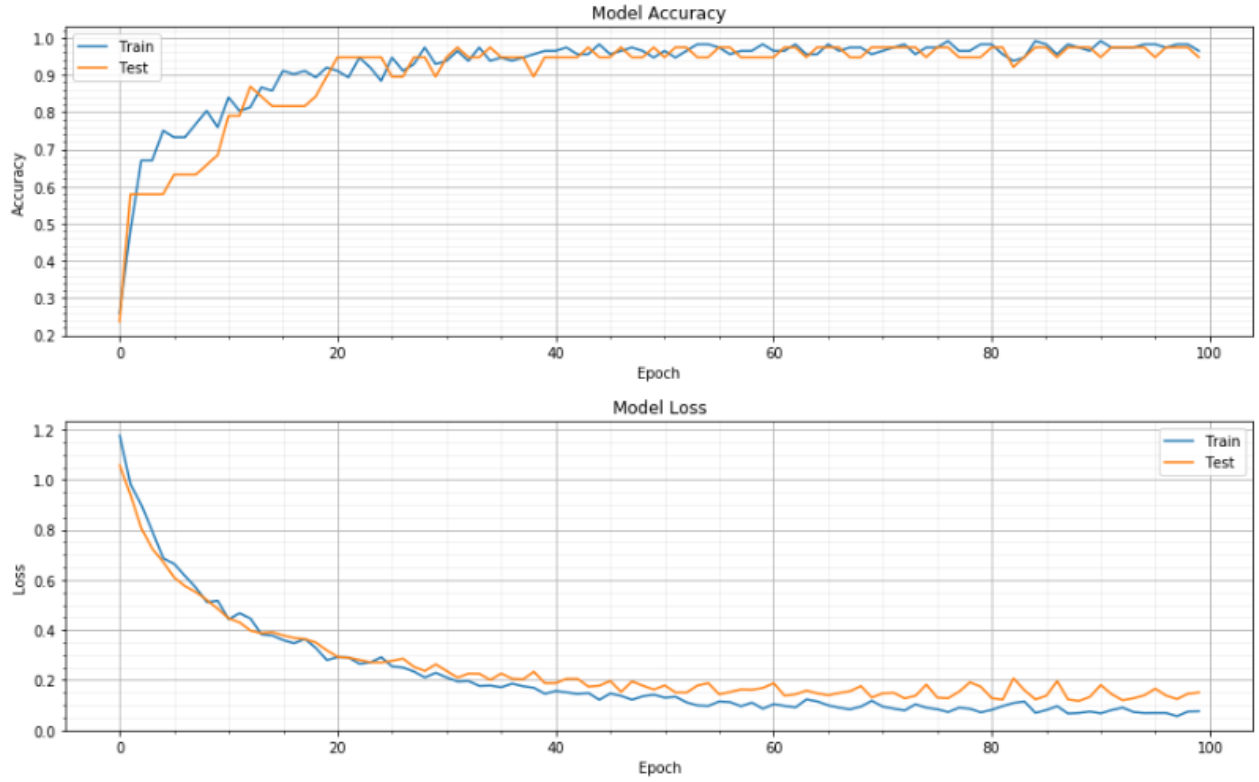
Fig. 2: model summary printout

(continued from previous page)

```
    model = fitted model
    loss_acc = input 'loss' or 'acc' to plot respective graph
    '''
    history = model.history.history

    if loss_acc == 'loss':
        axis_title = 'loss'
        title = 'Loss'
        epoch = len(history['loss'])
    elif loss_acc == 'acc':
        axis_title = 'acc'
        title = 'Accuracy'
        epoch = len(history['loss'])

    plt.figure(figsize=(15,4))
    plt.plot(history[axis_title])
    plt.plot(history['val_' + axis_title])
    plt.title('Model ' + title)
    plt.ylabel(title)
    plt.xlabel('Epoch')

    plt.grid(b=True, which='major')
    plt.minorticks_on()
    plt.grid(b=True, which='minor', alpha=0.2)

    plt.legend(['Train', 'Test'])
    plt.show()


plot_validate(model, 'acc', epoch)
plot_validate(model, 'loss', epoch)
```

12.1.4 Auto-Tuning

Unlike grid-search we can use Bayesian optimization for a faster hyperparameter tuning.



https://www.dlology.com/blog/how-to-do-hyperparameter-search-with-baysian-optimization-for-keras-model/ https://medium.com/@crawftv/parameter-hyperparameter-tuning-with-bayesian-optimization-7acf42d348e1

12.2 Model Compiling

12.2.1 Activation Functions

Input & Hidden Layers

ReLu (Rectified Linear units) is very popular compared to the now mostly obsolete sigmoid & tanh functions because it avoids vanishing gradient problem and has faster convergence. However, ReLu can only be used in hidden layers. Also, some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. Simply saying that ReLu could result in Dead Neurons.

To fix this problem another modification was introduced called Leaky ReLu to fix the problem of dying neurons. It introduces a small slope to keep the updates alive. We then have another variant made form both ReLu and Leaky ReLu called Maxout function .

Output Layer

Activation function

- Binary Classification: Sigmoid
- Multi-Class Classification: Softmax
- Regression: Linear

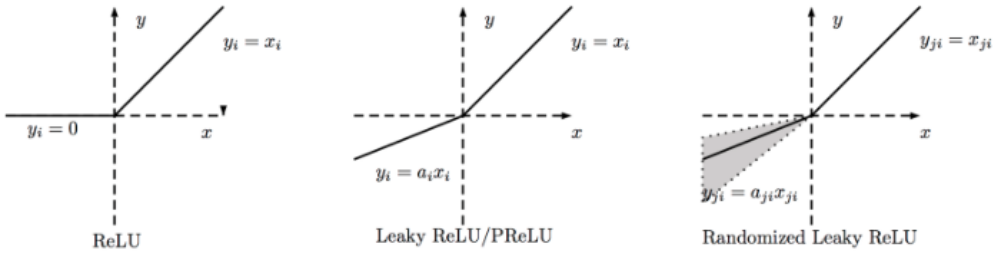


Fig. 3: https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f

12.2.2 Gradient Descent

Backpropagation, short for "backward propagation of errors," is an algorithm for supervised learning of artificial neural networks using gradient descent.

- **Optimizer** is a learning algorithm called gradient descent, refers to the calculation of an error gradient or slope of error and "descent" refers to the moving down along that slope towards some minimum level of error.
- **Batch Size** is a hyperparameter of gradient descent that controls the number of training samples to work through before the model's internal parameters are updated.
- **Epoch** is a hyperparameter of gradient descent that controls the number of complete passes through the training dataset.

Optimizers is used to find the minimium value of the cost function to perform backward propagation. There are more advanced adaptive optimizers, like AdaGrad/RMSprop/Adam, that allow the learning rate to adapt to the size of the gradient. The hyperparameters are essential to get the model to perform well.

Algo	Trick
SGD	
Momentum	Smooth updates
Nesterov	Interim calc gradient + Smooth
AdaGrad	Adaptive correction using Squared Gradient
RMSprop	EWMA applied to squared gradient adagrad
Adam	Adaptive, use EWMA on 1st and 2nd moments

Fig. 4: From Udemy, Zero to Hero Deep Learning with Python & Keras

Assume you have a dataset with 200 samples (rows of data) and you choose a batch size of 5 and 1,000 epochs. This means that the dataset will be divided into 40 batches, each with 5 samples. The model weights will be updated after each batch of 5 samples. This also means that one epoch will involve 40 batches or 40 updates to the model.

More here:

- https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.
- https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/
- https://blog.usejournal.com/stock-market-prediction-by-recurrent-neural-network-on-lstm-model-56de700bff68

12.3 ANN

12.3.1 Theory

An **artifical neural network** is the most basic form of neural network. It consists of an input layer, hidden layers, and an output layer. This writeup by Berkeley gave an excellent introduction to the theory. Most of the diagrams are taken from the site.

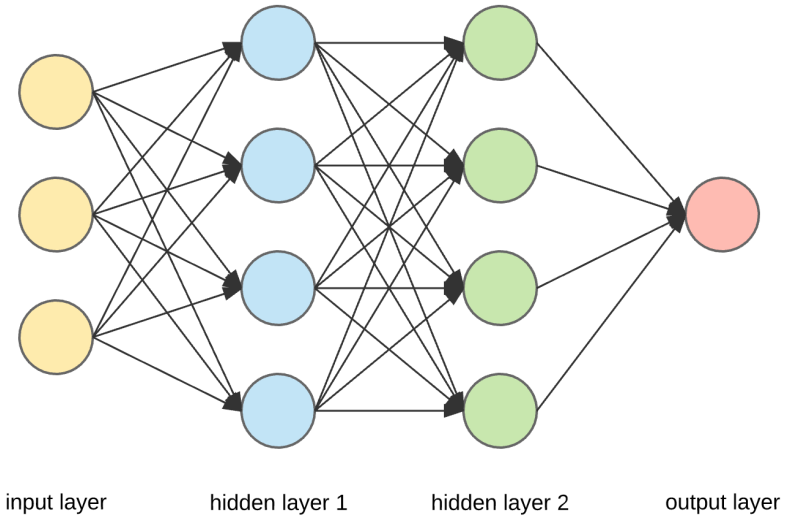


Fig. 5: Structure of an artificial neutral network

Zooming in at a single perceptron, the input layer consists of every individual features, each with an assigned weight feeding to the hidden layer. An **activation function** tells the perception what outcome it is.

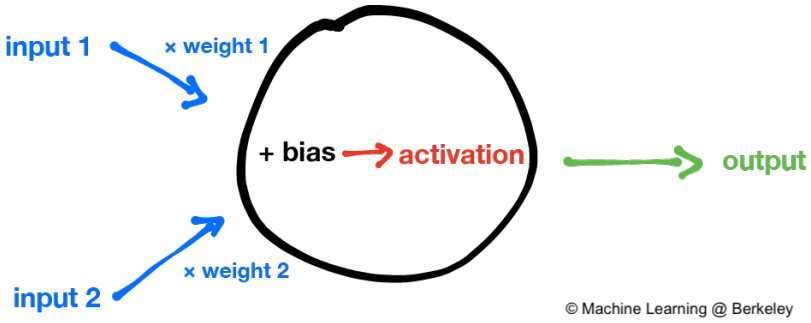


Fig. 6: Structure of a single perceptron

Activation functions consists of *ReLU*, *Tanh*, *Linear*, *Sigmoid*, *Softmax* and many others. Sigmoid is used for binary classifications, while softmax is used for multi-class classifications.

The backward propagation algorithm works in such that the slopes of gradient descent is calculated by working backwards from the output layer back to the input layer. The weights are readjusted to reduce the loss and improve the accuracy of the model.

A summary is as follows

1. Randomly initialize the weights for all the nodes.

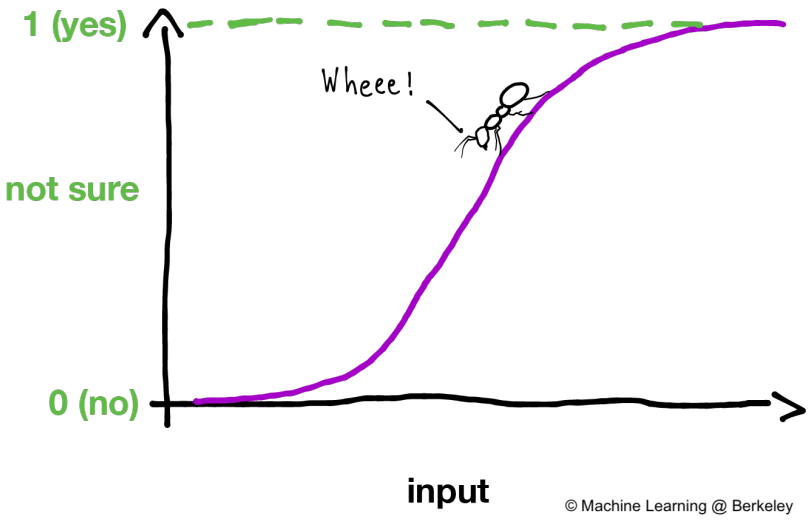


Fig. 7: An activation function, using sigmoid function

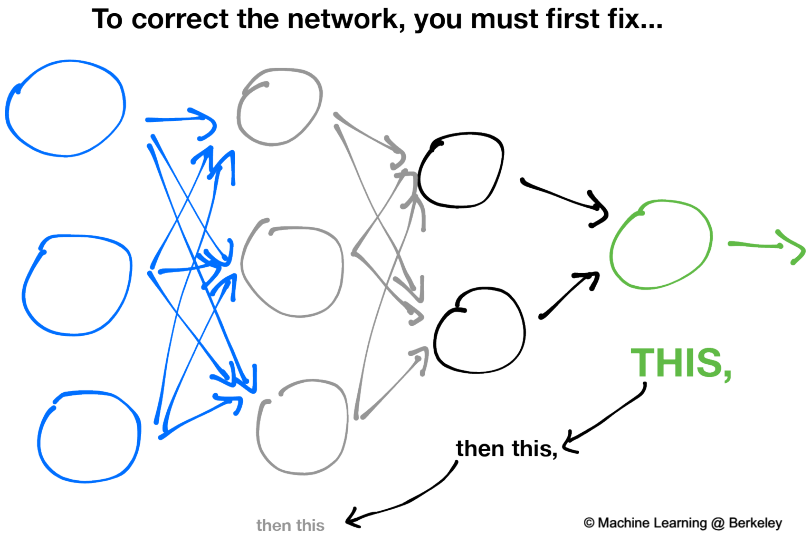


Fig. 8: Backward propagation

2. For every training example, perform a forward pass using the current weights, and calculate the output of each node going from left to right. The final output is the value of the last node.
3. Compare the final output with the actual target in the training data, and measure the error using a loss function.
4. Perform a backwards pass from right to left and propagate the error to every individual node using backpropagation. Calculate each weight's contribution to the error, and adjust the weights accordingly using gradient descent. Propagate the error gradients back starting from the last layer.

12.3.2 Keras Model

Building an ANN model in Keras library requires

- input & hidden layers
- model compliation
- model fitting
- model evalution

Definition of layers are typically done using the typical Dense layer, or regularization layer called Dropout. The latter prevents overfitting as it randomly selects neurons to be ignored during training.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# using dropout layers
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

Before training, the model needs to be compiled with the learning hyperparameters of optimizer, loss, and metric functions.

```
# from keras documentation
# https://keras.io/getting-started/sequential-model-guide/

# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')


# we can also set optimizer's parameters
from tensorflow.keras.optimizers import RMSprop
```

(continues on next page)

(continued from previous page)

```
rmsprop = RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
model.compile(optimizer=rmsprop, loss='mse')
```

We can also use sklearn's **cross-validation**.

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential

def create_model():
    model = Sequential()
    model.add(Dense(6, input_dim=4, kernel_initializer='normal', activation='relu'))
    #model.add(Dense(4, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

from sklearn.model_selection import cross_val_score
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Wrap our Keras model in an estimator compatible with scikit_learn
estimator = KerasClassifier(build_fn=create_model, epochs=100, verbose=0)
cv_scores = cross_val_score(estimator, all_features_scaled, all_classes, cv=10)
cv_scores.mean()
```

The below gives a compiled code example code.

```
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop

(mnist_train_images, mnist_train_labels), (mnist_test_images, mnist_test_labels) =␣
↪mnist.load_data()

train_images = mnist_train_images.reshape(60000, 784)
test_images = mnist_test_images.reshape(10000, 784)
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')
train_images /= 255
test_images /= 255

# convert the 0-9 labels into "one-hot" format, as we did for TensorFlow.
train_labels = keras.utils.to_categorical(mnist_train_labels, 10)
test_labels = keras.utils.to_categorical(mnist_test_labels, 10)


model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
model.summary()


Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               401920
_________________________________________________________________
```

(continues on next page)

(continued from previous page)

```
dense_1 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])


history = model.fit(train_images, train_labels,
                    batch_size=100, #no of samples per gradient update
                    epochs=10, #iteration
                    verbose=1, #0=no printout, 1=progress bar, 2=step-by-step printout
                    validation_data=(test_images, test_labels))

# Train on 60000 samples, validate on 10000 samples
# Epoch 1/10
# - 4s - loss: 0.2459 - acc: 0.9276 - val_loss: 0.1298 - val_acc: 0.9606
# Epoch 2/10
# - 4s - loss: 0.0991 - acc: 0.9700 - val_loss: 0.0838 - val_acc: 0.9733
# Epoch 3/10
# - 4s - loss: 0.0656 - acc: 0.9804 - val_loss: 0.0738 - val_acc: 0.9784
# Epoch 4/10
# - 4s - loss: 0.0493 - acc: 0.9850 - val_loss: 0.0650 - val_acc: 0.9798
# Epoch 5/10
# - 4s - loss: 0.0367 - acc: 0.9890 - val_loss: 0.0617 - val_acc: 0.9817
# Epoch 6/10
# - 4s - loss: 0.0281 - acc: 0.9915 - val_loss: 0.0698 - val_acc: 0.9800
# Epoch 7/10
# - 4s - loss: 0.0221 - acc: 0.9936 - val_loss: 0.0665 - val_acc: 0.9814
# Epoch 8/10
# - 4s - loss: 0.0172 - acc: 0.9954 - val_loss: 0.0663 - val_acc: 0.9823
# Epoch 9/10
# - 4s - loss: 0.0128 - acc: 0.9964 - val_loss: 0.0747 - val_acc: 0.9825
# Epoch 10/10
# - 4s - loss: 0.0098 - acc: 0.9972 - val_loss: 0.0840 - val_acc: 0.9795



score = model.evaluate(test_images, test_labels, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Here's another example using the Iris dataset.

```
import pandas as pd
import numpy as np

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
def modeling(X_train, y_train, X_test, y_test, features, classes, epoch, batch,␣
↪verbose, dropout):

    model = Sequential()

    #first layer input dim as number of features
    model.add(Dense(100, activation='relu', input_dim=features))
    model.add(Dropout(dropout))
    model.add(Dense(50, activation='relu'))
    #nodes must be same as no. of labels classes
    model.add(Dense(classes, activation='softmax'))

    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    model.fit(X_train, y_train,
            batch_size=batch,
            epochs= epoch,
            verbose=verbose,
            validation_data=(X_test, y_test))

    return model

iris = load_iris()
X = pd.DataFrame(iris['data'], columns=iris['feature_names'])
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X,y,random_state=0)

# define ANN model parameters
features = X_train.shape[1]
classes = len(np.unique(y_train))
epoch = 100
batch = 25
verbose = 0
dropout = 0.2

model = modeling(X_train, y_train, X_test, y_test, features, classes, epoch, batch,␣
↪verbose, dropout)
```

12.4 CNN

Convolutional Neural Network (CNN) is suitable for unstructured data like image classification, machine translation, sentence classification, and sentiment analysis.

12.4.1 Theory

This article from medium gives a good introduction of CNN. The steps goes something like this:

1. Provide input image into **convolution layer**
2. Choose parameters, apply filters with **strides**, **padding** if requires. Perform convolution on the image and apply **ReLU** activation to the matrix.

3. Perform **pooling** to reduce dimensionality size. Max-pooling is most commonly used
4. Add as many convolutional layers until satisfied
5. **Flatten** the output and feed into a fully connected layer (**FC Layer**)
6. Output the class using an activation function (Logistic Regression with cost functions) and classifies images.

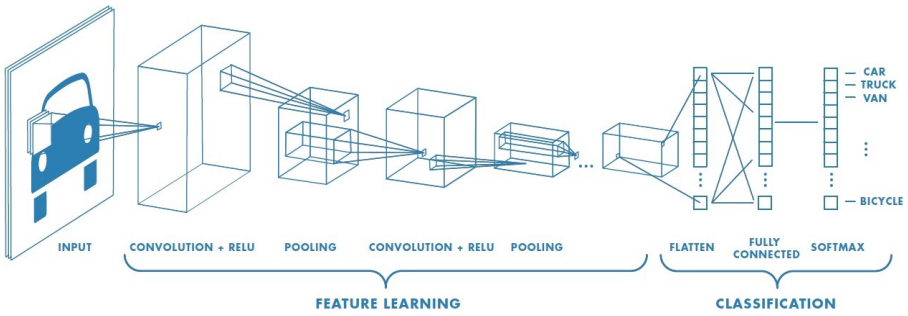


Fig. 9: from medium

There are many topologies, or CNN architecture to build on as the hyperparameters, layers etc. are endless. Some specialized architecture includes **LeNet-5** (handwriting recognition), **AlexNet** (deeper than LeNet, image classification), **GoogLeNet** (deeper than AlexNet, includes inception modules, or groups of convolution), **ResNet** (even deeper, maintains performance using skip connections). This article1 gives a good summary of each architecture.

12.4.2 Keras Model

```
import tensorflow
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import RMSprop

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))

# 64 3x3 kernels
model.add(Conv2D(64, (3, 3), activation='relu'))
# Reduce by taking the max of each 2x2 block
model.add(MaxPooling2D(pool_size=(2, 2)))
# Dropout to avoid overfitting
model.add(Dropout(0.25))
# Flatten the results to one dimension for passing into our final layer
model.add(Flatten())
# A hidden layer to learn with
model.add(Dense(128, activation='relu'))
# Another dropout
model.add(Dropout(0.5))
# Final categorization from 0-9 with softmax
model.add(Dense(10, activation='softmax'))

model.summary()
```

(continues on next page)

(continued from previous page)

```
# _________________________________________________________________
# Layer (type)                 Output Shape              Param #
# =================================================================
# conv2d (Conv2D)              (None, 26, 26, 32)        320
# _________________________________________________________________
# conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496
# _________________________________________________________________
# max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
# _________________________________________________________________
# dropout (Dropout)            (None, 12, 12, 64)        0
# _________________________________________________________________
# flatten (Flatten)            (None, 9216)              0
# _________________________________________________________________
# dense (Dense)                (None, 128)               1179776
# _________________________________________________________________
# dropout_1 (Dropout)          (None, 128)               0
# _________________________________________________________________
# dense_1 (Dense)              (None, 10)                1290
# =================================================================
# Total params: 1,199,882
# Trainable params: 1,199,882
# Non-trainable params: 0
# _________________________________________________________________

model.compile(loss='categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])

history = model.fit(train_images, train_labels,
                    batch_size=32,
                    epochs=10,
                    verbose=1,
                    validation_data=(test_images, test_labels))

# Train on 60000 samples, validate on 10000 samples
# Epoch 1/10
#  - 1026s - loss: 0.1926 - acc: 0.9418 - val_loss: 0.0499 - val_acc: 0.9834
# Epoch 2/10
#  - 995s - loss: 0.0817 - acc: 0.9759 - val_loss: 0.0397 - val_acc: 0.9874
# Epoch 3/10
#  - 996s - loss: 0.0633 - acc: 0.9811 - val_loss: 0.0339 - val_acc: 0.9895
# Epoch 4/10
#  - 991s - loss: 0.0518 - acc: 0.9836 - val_loss: 0.0302 - val_acc: 0.9909
# Epoch 5/10
#  - 996s - loss: 0.0442 - acc: 0.9861 - val_loss: 0.0322 - val_acc: 0.9905
# Epoch 6/10
#  - 994s - loss: 0.0395 - acc: 0.9878 - val_loss: 0.0303 - val_acc: 0.9898
# Epoch 7/10
#  - 1001s - loss: 0.0329 - acc: 0.9890 - val_loss: 0.0328 - val_acc: 0.9907
# Epoch 8/10
#  - 993s - loss: 0.0298 - acc: 0.9907 - val_loss: 0.0336 - val_acc: 0.9916
# Epoch 9/10
#  - 998s - loss: 0.0296 - acc: 0.9911 - val_loss: 0.0281 - val_acc: 0.9915
# Epoch 10/10
#  - 996s - loss: 0.0252 - acc: 0.9917 - val_loss: 0.0340 - val_acc: 0.9918

score = model.evaluate(test_images, test_labels, verbose=0)
```

(continues on next page)

(continued from previous page)

```
print('Test loss:', score[0])
print('Test accuracy:', score[1])

# Test loss: 0.034049834153382426
# Test accuracy: 0.9918
```

12.5 RNN

Recurrent Neural Network (RNN). A typical RNN looks like below, where X(t) is input, h(t) is output and A is the neural network which gains information from the previous step in a loop. The output of one unit goes into the next one and the information is passed.

12.5.1 Theory

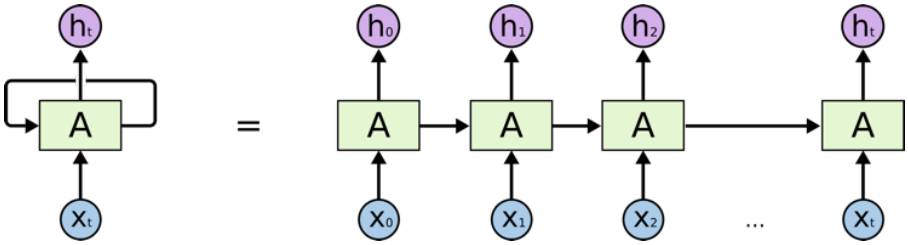


Fig. 10: from medium

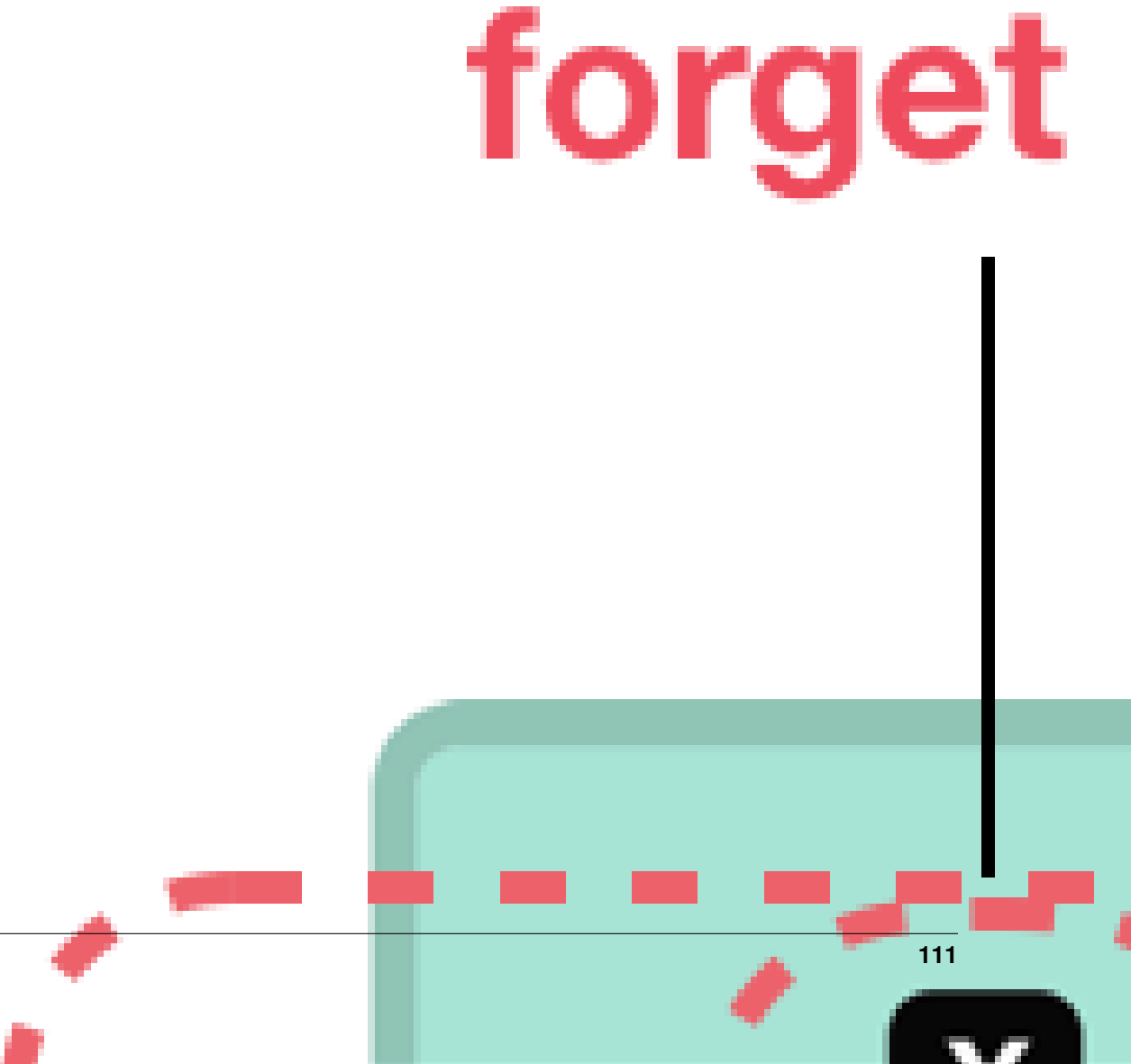
Long Short Term Memory (LSTM) is a special kind of Recurrent Neural Networks (RNN) with the capability of learning long-term dependencies. The intricacies lie within the cell, where 3 internal mechanisms called gates regulate the flow of information. This consists of 4 activation functions, 3 sigmoid and 1 tanh, instead of the typical 1 activation function. This medium from article gives a good description of it. An alternative, or simplified form of LSTM is **Gated Recurrent Unit (GRU)**.

12.5.2 Keras Model

LSTM requires input needs to be of shape `(num_sample, time_steps, num_features)` if using tensorflow backend. This can be processed using keras's `TimeseriesGenerator`.

```
from keras.preprocessing.sequence import TimeseriesGenerator

### UNIVARIATE --------------------
time_steps = 6
sampling_rate = 1
num_sample = 4

X = [1,2,3,4,5,6,7,8,9,10]
y = [5,6,7,8,9,1,2,3,4,5]

data = TimeseriesGenerator(X, y,
                           length=time_steps,
                           sampling_rate=sampling_rate,
                           batch_size=num_sample)
```

(continues on next page)


forget

(continued from previous page)

```
data[0]

# (array([[1, 2, 3, 4, 5, 6],
#         [2, 3, 4, 5, 6, 7],
#         [3, 4, 5, 6, 7, 8],
#         [4, 5, 6, 7, 8, 9]]), array([2, 3, 4, 5]))
# note that y-label is the next time step away


### MULTIVARIATE ---------------------
# from pandas df
df = pd.DataFrame(np.random.randint(1, 5, (10,3)), columns=['col1','col2','label'])
X = df[['col1','col2']].values
y = df['label'].values

time_steps = 6
sampling_rate = 1
num_sample = 4

data = TimeseriesGenerator(X, y,
                           length=time_steps,
                           sampling_rate=sampling_rate,
                           batch_size=num_sample)

X = data[0][0]
y = data[0][1]
```

The code below uses LSTM for sentiment analysis in IMDB movie reviews.

```
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding
from tensorflow.keras.layers import LSTM
from tensorflow.keras.datasets import imdb

# words in sentences are encoded into integers
# response is in binary 1-0
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=20000)

# limit the sentence to backpropagate back 80 words through time
x_train = sequence.pad_sequences(x_train, maxlen=80)
x_test = sequence.pad_sequences(x_test, maxlen=80)


# embedding layer converts input data into dense vectors of fixed size of 20k words &␣
↪128 hidden neurons, better suited for neural network
model = Sequential()
model.add(Embedding(20000, 128)) #for nlp
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2)) #128 memory cells
model.add(Dense(1, activation='sigmoid')) #1 class classification, sigmoid for binary␣
↪classification

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(x_train, y_train,
```

(continues on next page)

(continued from previous page)

```
        batch_size=32,
        epochs=15,
        verbose=1,
        validation_data=(x_test, y_test))

# Train on 25000 samples, validate on 25000 samples
# Epoch 1/15
#  - 139s - loss: 0.6580 - acc: 0.5869 - val_loss: 0.5437 - val_acc: 0.7200
# Epoch 2/15
#  - 138s - loss: 0.4652 - acc: 0.7772 - val_loss: 0.4024 - val_acc: 0.8153
# Epoch 3/15
#  - 136s - loss: 0.3578 - acc: 0.8446 - val_loss: 0.4024 - val_acc: 0.8172
# Epoch 4/15
#  - 134s - loss: 0.2902 - acc: 0.8784 - val_loss: 0.3875 - val_acc: 0.8276
# Epoch 5/15
#  - 135s - loss: 0.2342 - acc: 0.9055 - val_loss: 0.4063 - val_acc: 0.8308
# Epoch 6/15
#  - 132s - loss: 0.1818 - acc: 0.9292 - val_loss: 0.4571 - val_acc: 0.8308
# Epoch 7/15
#  - 124s - loss: 0.1394 - acc: 0.9476 - val_loss: 0.5458 - val_acc: 0.8177
# Epoch 8/15
#  - 126s - loss: 0.1062 - acc: 0.9609 - val_loss: 0.5950 - val_acc: 0.8133
# Epoch 9/15
#  - 133s - loss: 0.0814 - acc: 0.9712 - val_loss: 0.6440 - val_acc: 0.8218
# Epoch 10/15
#  - 134s - loss: 0.0628 - acc: 0.9783 - val_loss: 0.6525 - val_acc: 0.8138
# Epoch 11/15
#  - 136s - loss: 0.0514 - acc: 0.9822 - val_loss: 0.7252 - val_acc: 0.8143
# Epoch 12/15
#  - 137s - loss: 0.0414 - acc: 0.9869 - val_loss: 0.7997 - val_acc: 0.8035
# Epoch 13/15
#  - 136s - loss: 0.0322 - acc: 0.9890 - val_loss: 0.8717 - val_acc: 0.8120
# Epoch 14/15
#  - 132s - loss: 0.0279 - acc: 0.9905 - val_loss: 0.9776 - val_acc: 0.8114
# Epoch 15/15
#  - 140s - loss: 0.0231 - acc: 0.9918 - val_loss: 0.9317 - val_acc: 0.8090
# Out[8]:
# <tensorflow.python.keras.callbacks.History at 0x21c29ab8630>

score, acc = model.evaluate(x_test, y_test,
                            batch_size=32,
                            verbose=1)
print('Test score:', score)
print('Test accuracy:', acc)

# Test score: 0.9316869865119457
# Test accuracy: 0.80904
```

This example uses a stock daily output for prediction.

```
from tensorflow.keras.preprocessing import sequence
from keras.preprocessing.sequence import TimeseriesGenerator

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding
from tensorflow.keras.layers import LSTM, GRU
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import pandas_datareader.data as web
from datetime import datetime

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

def stock(code, years_back):
    end = datetime.now()
    start = datetime(end.year-years_back, end.month, end.day)
    code = '{}.SI'.format(code)
    df = web.DataReader(code, 'yahoo', start, end)
    return df


def lstm(X_train, y_train, X_test, y_test, classes, epoch, batch, verbose, dropout)
    model = Sequential()
    # return sequences refer to all the outputs of the memory cells, True if next␣
↪layer is LSTM
    model.add(LSTM(50, dropout=dropout, recurrent_dropout=0.2, return_sequences=True,␣
↪input_shape=X.shape[1:]))
    model.add(LSTM(50, dropout=dropout, recurrent_dropout=0.2, return_
↪sequences=False))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    model.fit(X, y,
              batch_size=batch,
              epochs= epoch,
              verbose=verbose,
              validation_data=(X_test, y_test))
    return model


df = stock('S68', 10)

# train-test split-------------
df1 = df[:2400]
df2 = df[2400:]

X_train = df1[['High','Low','Open','Close','Volume']].values
y_train = df1['change'].values
X_test = df2[['High','Low','Open','Close','Volume']].values
y_test = df2['change'].values

# normalisation-------------
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Conversion to keras LSTM data format-------------
time_steps = 10
sampling_rate = 1
num_sample = 1200
```

(continues on next page)

(continued from previous page)

```
data = TimeseriesGenerator(X, y,
                           length=time_steps,
                           sampling_rate=sampling_rate,
                           batch_size=num_sample)
X_train = data[0][0]
y_train = data[0][1]

data = TimeseriesGenerator(X_test, y_test,
                           length=time_steps,
                           sampling_rate=sampling_rate,
                           batch_size=num_sample)
X_test = data[0][0]
y_test = data[0][1]


# model validation-------------
classes = 1
epoch = 2000
batch = 200
verbose = 0
dropout = 0.2

model = lstm(X_train, y_train, X_test, y_test, classes, epoch, batch, verbose,␣
↪dropout)

# draw loss graph
plot_validate(model, 'loss')


# draw train & test prediction
predict_train = model.predict(X_train)
predict_test = model.predict(X_test)

for real, predict in [(y_train, predict_train),(y_test, predict_test)]:
    plt.figure(figsize=(15,4))
    plt.plot(real)
    plt.plot(predict)
    plt.ylabel('Close Price');
    plt.legend(['Real', 'Predict']);
```

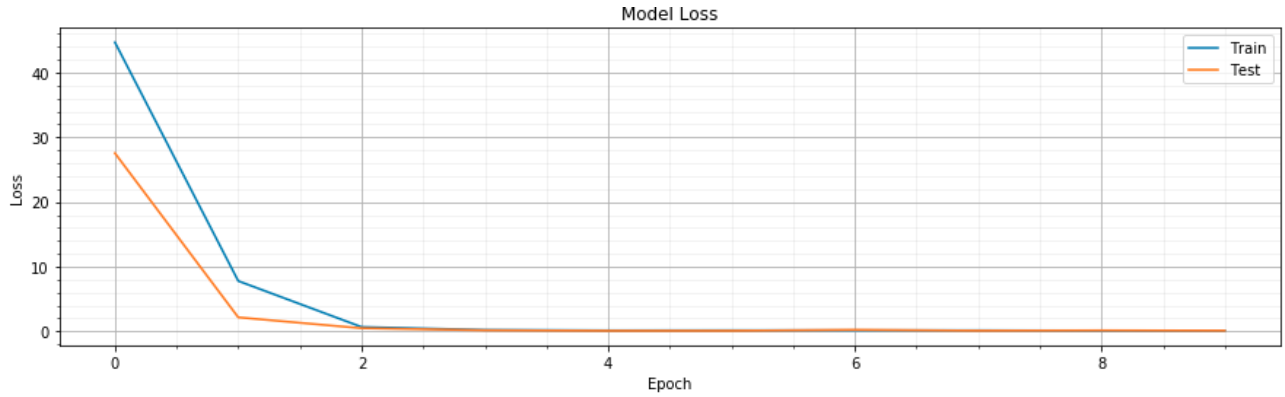


Fig. 12: Loss graph

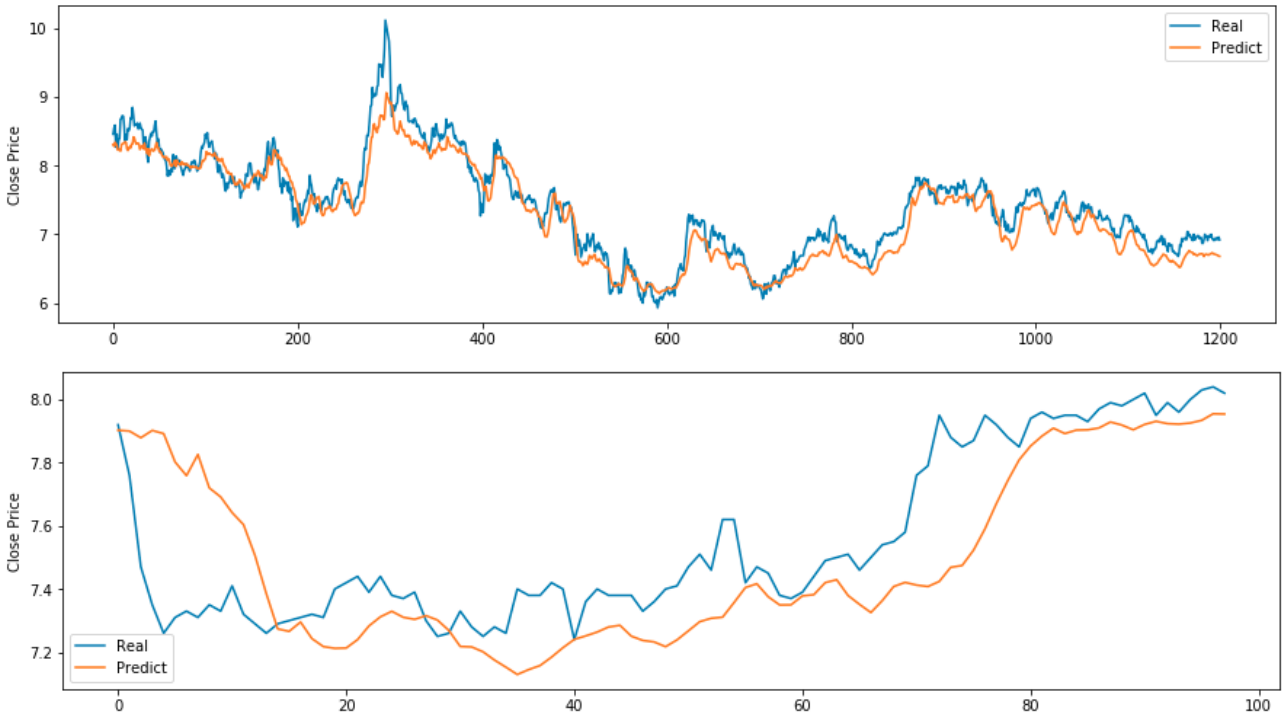


Fig. 13: Prediction graphs

12.6 Saving the Model

From Keras documentation, it is not recommended to save the model in a pickle format. Keras allows saving in a HDF5 format. This saves the entire model architecture, weights and optimizers.

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

To save just the architecture, see https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model.

CHAPTER 13

Reinforcement Learning

Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.

13.1 Concepts

13.1.1 Elements of Reinforcement Learning

Basic Elements

Term	Description
Agent	A model/algorithm that is taked with learning to accomplish a task
Environment	The world where agent acts in.
Action	A decision the agent makes in an environment
Reward Signal	A scalar indication of how well the agent is performing a task
State	A description of the environment that can be perceived by the agent
Terminal State	A state at which no further actions can be made by an agent

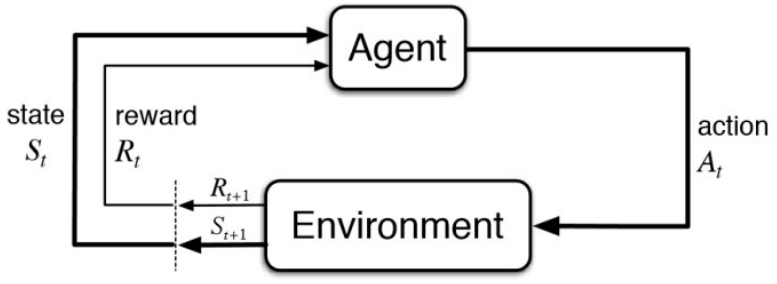


Fig. 1: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html

Algorithms for the Agent

Term	Description
Policy (π)	Function that outputs decisions the agent makes. In simple terms, it instructs what the agent should do at each state.
Value Function	Function that describes how good or bad a state is. It is the total amount of reward an agent is predicted to accumulate over the future, starting from a state.
Model of Environment	Predicts how the environment will reac tto the agent's actions. In given a state & action, what is the next state and reward. Such an approach is called a model-based method, in contrast with model-free methods.

13.1.2 Markov Decision Process

Reinforcement learning helps to solve Markov Decision Process (MDP). The core problem of MDPs is to find a "policy" for the decision maker: a function π that specifies the action $\pi(s)$ that the decision maker will choose when in state s. The diagram illustrate the Markov Decision Process.

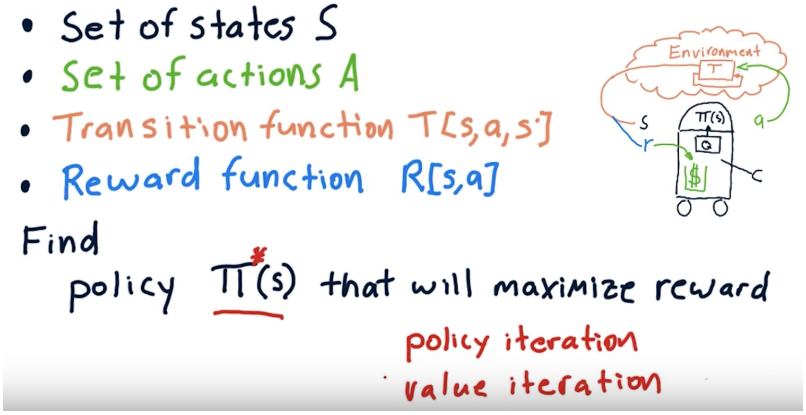


Fig. 2: Udacity, Machine Learning for Trading

13.2 Q-Learning

Q-Learning is an example of model-free reinforcement learning to solve the Markov Decision Process. It derives the policy by directly looking at the data instead of developing a model.

We first build a Q-table with each column as the type of action possible, and then each row as the number of possible states. And initialise the table with all zeros.

Updating the function Q uses the following Bellman equation. Algorithms using such equation as an iterative update are called `value iteration` algorithms.

Learning Hyperparameters

- **Learning Rate** (α): how quickly a network abandons the former value for the new. If the learning rate is 1, the new estimate will be the new Q-value.
- **Discount Rate** (γ): how much to discount the future reward. The idea is that the later a reward comes, the less valuable it becomes. Think inflation of money in the real world.

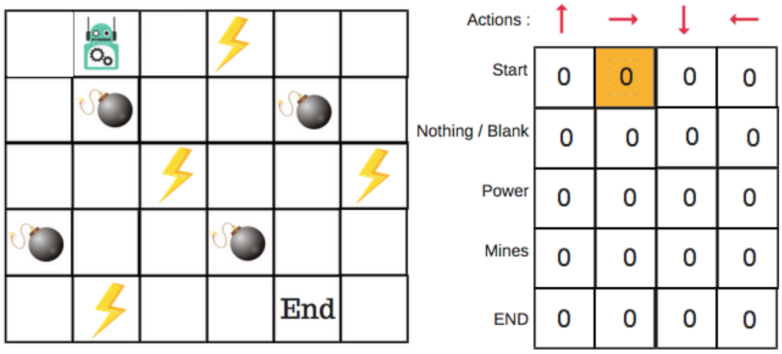


Fig. 3: from Medium

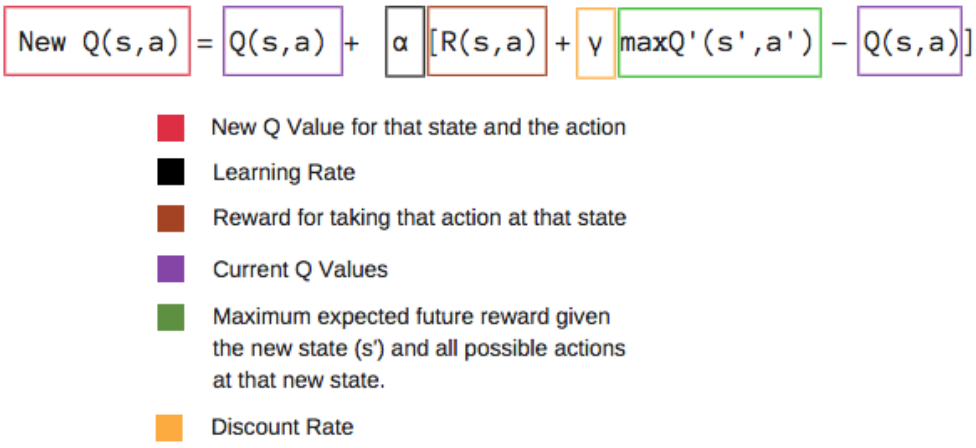


Fig. 4: from Medium

Exploration vs Exploitation

A central dilemma of reinforcement learning is to *exploit* what it has already experienced in order to obtain a reward. But in order to do that, it has to *explore* in order to make better actions in the future.

This is known as the epsilon greedy strategy. In the beginning, the epsilon rates will be higher. The bot will explore the environment and randomly choose actions. The logic behind this is that the bot does not know anything about the environment. However the more the bot explores the environment, the more the epsilon rate will decreases and the bot starts to exploit the environment.

There are other algothrims to manage the exploration vs exploiation problem, like softmax.

Definitions

- **argmax(x)**: position where the first max value occurs

Code

Start the environment and training parameters for frozen lake in AI gym.

```
#code snippets from https://gist.github.com/simoninithomas/
↪baafe42d1a665fb297ca669aa2fa6f92#file-q-learning-with-frozenlake-ipynb

import numpy as np
import gym
import random

env = gym.make("FrozenLake-v0")

action_size = env.action_space.n
state_size = env.observation_space.n

qtable = np.zeros((state_size, action_size))
print(qtable)

# define hyperparameters ----------
total_episodes = 15000        # Total episodes
learning_rate = 0.8           # Learning rate
max_steps = 99                # Max steps per episode
gamma = 0.95                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.005             # Exponential decay rate for exploration prob
```

Train and generate the Q-table.

```
# generate Q-table ------------
# List of rewards
rewards = []

# 2 For life or until learning is stopped
for episode in range(total_episodes):
    # Reset the environment
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
```

(continues on next page)

(continued from previous page)

```
    for step in range(max_steps):
        # 3. Choose an action a in the current world state (s)
        ## First we randomize a number
        exp_exp_tradeoff = random.uniform(0, 1)

        ## If this number > greater than epsilon --> exploitation (taking the biggest␣
↪Q value for this state)
        if exp_exp_tradeoff > epsilon:
            action = np.argmax(qtable[state,:])

        # Else doing a random choice --> exploration
        else:
            action = env.action_space.sample()

        # Take the action (a) and observe the outcome state(s') and reward (r)
        new_state, reward, done, info = env.step(action)

        # Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
        # qtable[new_state,:] : all the actions we can take from new state
        qtable[state, action] = qtable[state, action] + learning_rate * (reward +␣
↪gamma * np.max(qtable[new_state, :]) - qtable[state, action])

        total_rewards += reward

        # Our new state is state
        state = new_state

        # If done (if we're dead) : finish episode
        if done == True:
            break

    # Reduce epsilon (because we need less and less exploration)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
    rewards.append(total_rewards)

print ("Score over time: " +  str(sum(rewards)/total_episodes))
print(qtable)
```

Rerun the game using the Q-table generated.

```
env.reset()

for episode in range(5):
    state = env.reset()
    step = 0
    done = False
    print("****************************************************")
    print("EPISODE ", episode)

    for step in range(max_steps):

        # Take the action (index) that have the maximum expected future reward given␣
↪that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)
```

(continues on next page)

(continued from previous page)

```
        if done:
            # Here, we decide to only print the last state (to see if our agent is on␣
↪the goal or fall into an hole)
            env.render()

            # We print the number of step it took.
            print("Number of steps", step)
            break
        state = new_state
env.close()
```

13.3 Resources

- https://towardsdatascience.com/reinforcement-learning-implement-grid-world-from-scratch-c5963765ebff
- https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-97
- https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-

CHAPTER 14

Evaluation

Sklearn provides a good list of evaluation metrics for classification, regression and clustering problems.

http://scikit-learn.org/stable/modules/model_evaluation.html

In addition, it is also essential to know how to analyse the features and adjusting hyperparameters based on different evalution metrics.

14.1 Classification

14.1.1 Confusion Matrix

		Actual class		
		Cat	**Dog**	**Rabbit**
Predicted class	**Cat**	5	2	0
	Dog	3	3	2
	Rabbit	0	1	11

Fig. 1: Wikipedia

Recall|Sensitivity: (True Positive / True Positive + False Negative) High recall means to get all positives (i.e., True Positive + False Negative) despite having some false positives. Search & extraction in legal cases, Tumour detection. Often need humans to filter false positives.

		Actual class	
		Cat	**Non-cat**
Predicted class	**Cat**	5 True Positives	2 False Positives
	Non-cat	3 False Negatives	17 True Negatives

Fig. 2: Wikipedia

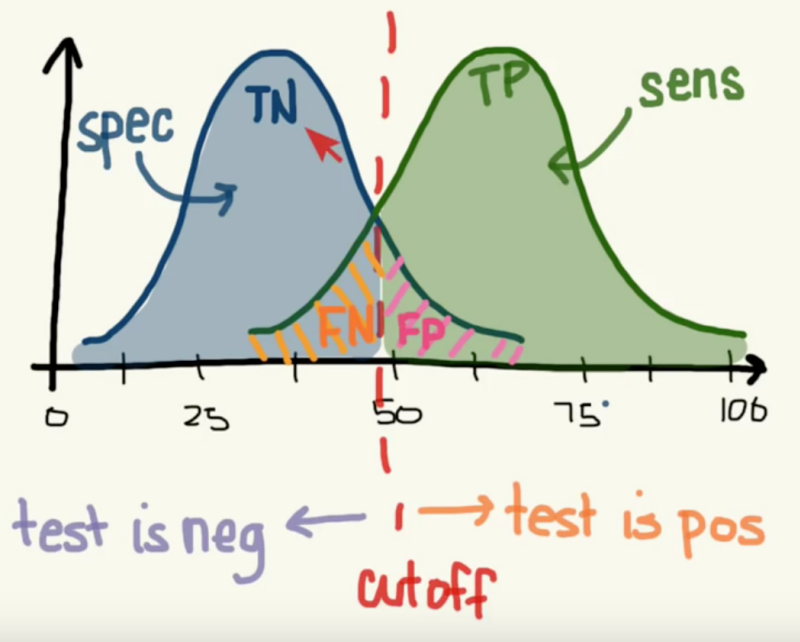


Fig. 3: https://www.youtube.com/watch?v=21Igj5Pr6u4

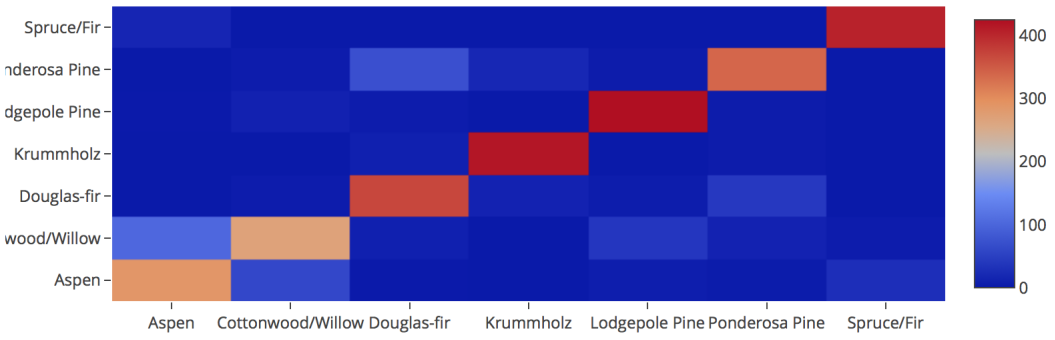
Precision: (True Positive / True Positive + False Positive) High precision means it is important to filter off the any false positives. Search query suggestion, Document classification, customer-facing tasks.

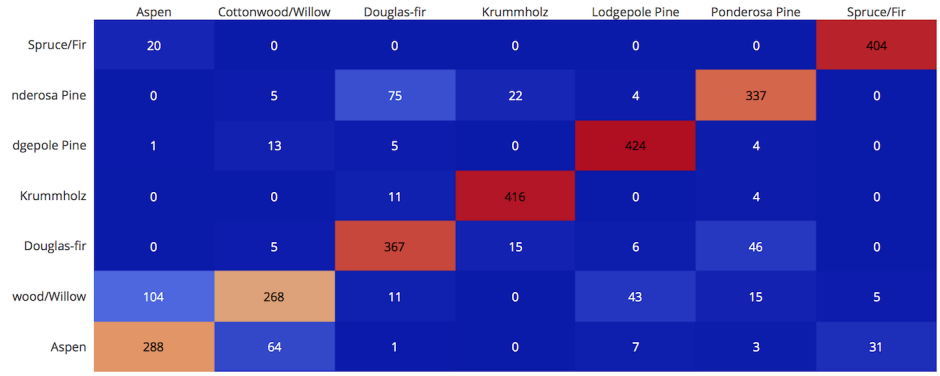
F1-Score: is the harmonic mean of precision and sensitivity, ie., 2*((precision * recall) / (precision + recall))

1. Confusion Matrix

Plain vanilla matrix. Not very useful as does not show the labels. However, the matrix can be used to build a heatmap using plotly directly.

```
print (sklearn.metrics.confusion_matrix(test_target,predictions))
array([[288,  64,   1,   0,   7,   3,  31],
       [104, 268,  11,   0,  43,  15,   5],
       [  0,   5, 367,  15,   6,  46,   0],
       [  0,   0,  11, 416,   0,   4,   0],
       [  1,  13,   5,   0, 424,   4,   0],
       [  0,   5,  75,  22,   4, 337,   0],
       [ 20,   0,   0,   0,   0,   0, 404]])

# make heatmap using plotly
from plotly.offline import iplot
from plotly.offline import init_notebook_mode
import plotly.graph_objs as go
init_notebook_mode(connected=True)

layout = go.Layout(width=800, height=400)
data = go.Heatmap(z=x,x=title,y=title)
fig = go.Figure(data=[data], layout=layout)
iplot(fig)

# this gives the values of each cell, but api unable to change the layout size
import plotly.figure_factory as ff
layout = go.Layout(width=800, height=500)
data = ff.create_annotated_heatmap(z=x,x=title,y=title)
iplot(data)
```





With pandas crosstab. Convert encoding into labels and put the two pandas series into a crosstab.

```
def forest(x):
  if x==1:
      return 'Spruce/Fir'
  elif x==2:
      return 'Lodgepole Pine'
  elif x==3:
      return 'Ponderosa Pine'
  elif x==4:
      return 'Cottonwood/Willow'
  elif x==5:
      return 'Aspen'
  elif x==6:
      return 'Douglas-fir'
  elif x==7:
      return 'Krummholz'

# Create pd Series for Original
# need to reset index as train_test is randomised
Original = test_target.apply(lambda x: forest(x)).reset_index(drop=True)
Original.name = 'Original'

# Create pd Series for Predicted
Predicted = pd.DataFrame(predictions, columns=['Predicted'])
Predicted = Predicted[Predicted.columns[0]].apply(lambda x: forest(x))

# Create Confusion Matrix
confusion = pd.crosstab(Original, Predicted)
confusion
```

Predicted	Aspen	Cottonwood/Willow	Douglas-fir	Krummholz	Lodgepole Pine	Ponderosa Pine	Spruce/Fir
Original							
Aspen	382	0	4	0	20	7	1
Cottonwood/Willow	0	408	5	0	0	6	0
Douglas-fir	8	16	342	0	5	60	0
Krummholz	2	0	0	421	4	0	19
Lodgepole Pine	30	0	10	7	304	10	103
Ponderosa Pine	2	19	52	0	3	344	0
Spruce/Fir	8	0	3	26	70	0	323

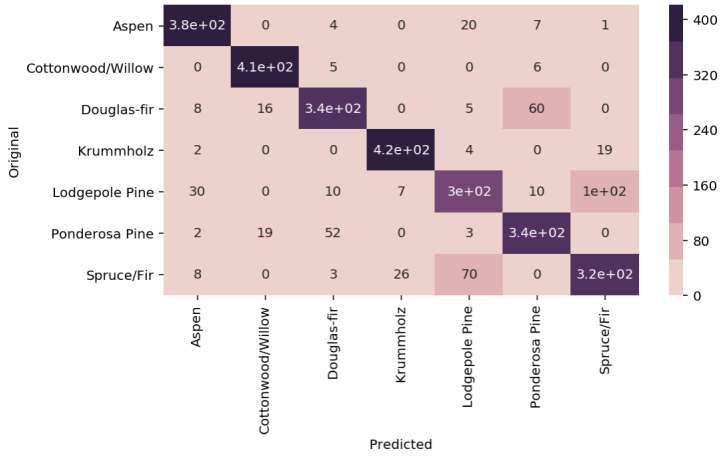
Using a heatmap.

```
# add confusion matrix from pd.crosstab earlier
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(10, 5))
sns.heatmap(confusion,annot=True,cmap=sns.cubehelix_palette(8));
```



2. Evaluation Metrics

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Accuracy = TP + TN / (TP + TN + FP + FN)
# Precision = TP / (TP + FP)
# Recall = TP / (TP + FN)  Also known as sensitivity, or True Positive Rate
# F1 = 2 * (Precision * Recall) / (Precision + Recall)

print('Accuracy:', accuracy_score(y_test, tree_predicted)
print('Precision:', precision_score(y_test, tree_predicted)
print('Recall:', recall_score(y_test, tree_predicted)
print('F1:', f1_score(y_test, tree_predicted)

Accuracy: 0.95
Precision: 0.79
Recall: 0.60
F1: 0.68

# for precision/recall/f1 in multi-class classification
# need to add average=None or will prompt an error
# scoring will be for each label, and averaging them is necessary
from statistics import mean
mean(f1_score(y_test, y_predict, average=None))
```

There are many other evaluation metrics, a list can be found here:

```
from sklearn.metrics.scorer import SCORERS

for i in sorted(list(SCORERS.keys())):
    print i

accuracy
adjusted_rand_score
average_precision
f1
f1_macro
```

(continues on next page)

(continued from previous page)

```
f1_micro
f1_samples
f1_weighted
log_loss
mean_absolute_error
mean_squared_error
median_absolute_error
neg_log_loss
neg_mean_absolute_error
neg_mean_squared_error
neg_median_absolute_error
precision
precision_macro
precision_micro
precision_samples
precision_weighted
r2
recall
recall_macro
recall_micro
recall_samples
recall_weighted
roc_auc
```

3. Classification Report

```
# Combined report with all above metrics
from sklearn.metrics import classification_report

print(classification_report(y_test, tree_predicted, target_names=['not 1', '1']))

             precision    recall  f1-score   support

      not 1       0.96      0.98      0.97       407
          1       0.79      0.60      0.68        43

avg / total       0.94      0.95      0.94       450
```

Classification report shows the details of precision, recall & f1-scores. It might be misleading to just print out a binary classification as their determination of True Positive, False Positive might differ from us. The report will tease out the details as shown below. We can also set `average=None` & compute the mean when printing out each individual scoring.

```
accuracy = accuracy_score(y_test, y_predict)
confusion = confusion_matrix(y_test,y_predict)
f1 = f1_score(y_test, y_predict)
recall = recall_score(y_test, y_predict)
precision = precision_score(y_test, y_predict)

f1_avg = mean(f1_score(y_test, y_predict, average=None))
recall_avg = mean(recall_score(y_test, y_predict, average=None))
precision_avg = mean(precision_score(y_test, y_predict, average=None))

print('accuracy:\t', accuracy)
print('\nf1:\t\t',f1)
print('recall\t\t',recall)
```

(continues on next page)

(continued from previous page)

```
print('precision\t',precision)

print('\nf1_avg:\t\t',f1_avg)
print('recall_avg\t',recall_avg)
print('precision_avg\t',precision_avg)

print('\nConfusion Matrix')
print(confusion)
print('\n',classification_report(y_test, y_predict))
```

```
accuracy:        0.8446153846153847

f1:              0.9157631359466223
recall           1.0
precision        0.8446153846153847

f1_avg:          0.45788156797331114
recall_avg       0.5
precision_avg    0.42230769230769233

Confusion Matrix
[[  0 101]
 [  0 549]]

              precision    recall  f1-score   support

           0       0.00      0.00      0.00       101
           1       0.84      1.00      0.92       549

   micro avg       0.84      0.84      0.84       650
   macro avg       0.42      0.50      0.46       650
weighted avg       0.71      0.84      0.77       650
```

Fig. 4: University of Michigan: Coursera Data Science in Python

4. Decision Function

```
X_train, X_test, y_train, y_test = train_test_split(X, y_binary_imbalanced, random_
↪state=0)
y_scores_lr = lr.fit(X_train, y_train).decision_function(X_test)
y_score_list = list(zip(y_test[0:20], y_scores_lr[0:20]))

# show the decision_function scores for first 20 instances
y_score_list

[(0, -23.176682692580048),
 (0, -13.541079101203881),
 (0, -21.722576315155052),
 (0, -18.90752748077151),
 (0, -19.735941639551616),
 (0, -9.7494967330877031),
 (1, 5.2346395208185506),
 (0, -19.307366394398947),
```

(continues on next page)

(continued from previous page)

```
 (0, -25.101037079396367),
 (0, -21.827003670866031),
 (0, -24.15099619980262),
 (0, -19.576751014363683),
 (0, -22.574837580426664),
 (0, -10.823683312193941),
 (0, -11.91254508661434),
 (0, -10.979579441354835),
 (1, 11.20593342976589),
 (0, -27.645821704614207),
 (0, -12.85921201890492),
 (0, -25.848618861971779)]
```

5. Probability Function

```
X_train, X_test, y_train, y_test = train_test_split(X, y_binary_imbalanced, random_
↪state=0)
# note that the first column of array indicates probability of predicting negative␣
↪class,
# 2nd column indicates probability of predicting positive class
y_proba_lr = lr.fit(X_train, y_train).predict_proba(X_test)
y_proba_list = list(zip(y_test[0:20], y_proba_lr[0:20,1]))

# show the probability of positive class for first 20 instances
y_proba_list
```

```
[(0, 8.5999236926158807e-11),
 (0, 1.31578065170999e-06),
 (0, 3.6813318939966053e-10),
 (0, 6.1456121155693793e-09),
 (0, 2.6840428788564424e-09),
 (0, 5.8320607398268079e-05),
 (1, 0.99469949997393026),
 (0, 4.1201906576825675e-09),
 (0, 1.2553305740618937e-11),
 (0, 3.3162918920398805e-10),
 (0, 3.2460530855408745e-11),
 (0, 3.1472051953481208e-09),
 (0, 1.5699022391384567e-10),
 (0, 1.9921654858205874e-05),
 (0, 6.7057057309326073e-06),
 (0, 1.704597440356912e-05),
 (1, 0.99998640688336282),
 (0, 9.8530840165646881e-13),
 (0, 2.6020404794341749e-06),
 (0, 5.9441185633886803e-12)]
```

14.1.2 Precision-Recall Curves

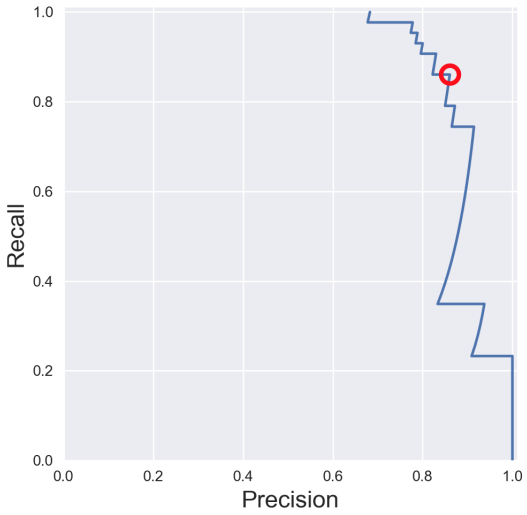
If your problem involves kind of searching a needle in the haystack; the positive class samples are very rare compared to the negative classes, use a precision recall curve.

```
from sklearn.metrics import precision_recall_curve

# get decision function scores
```

(continues on next page)

(continued from previous page)

```
y_scores_lr = m.fit(X_train, y_train).decision_function(X_test)

# get precision & recall values
precision, recall, thresholds = precision_recall_curve(y_test, y_scores_lr)
closest_zero = np.argmin(np.abs(thresholds))
closest_zero_p = precision[closest_zero]
closest_zero_r = recall[closest_zero]

plt.figure()
plt.xlim([0.0, 1.01])
plt.ylim([0.0, 1.01])
plt.plot(precision, recall, label='Precision-Recall Curve')
plt.plot(closest_zero_p, closest_zero_r, 'o', markersize = 12, fillstyle = 'none', c=
↪'r', mew=3)
plt.xlabel('Precision', fontsize=16)
plt.ylabel('Recall', fontsize=16)
plt.axes().set_aspect('equal')
plt.show()
```



14.1.3 ROC Curves

Receiver Operating Characteristic (ROC) is used to show the performance of a binary classifier. Y-axis is True Positive Rate (Recall) & X-axis is False Positive Rate (Fall-Out). Area Under Curve (AUC) of a ROC is used. Higher AUC better.

The term came about in WWII where this metrics is used to determined a receiver operator's ability to distinguish false positive and true postive correctly in the radar signals.

Some classifiers have a decision_function method while others have a probability prediction method, and some have both. Whichever one is available works fine for an ROC curve.

```
from sklearn.metrics import roc_curve, auc

X_train, X_test, y_train, y_test = train_test_split(X, y_binary_imbalanced, random_
↪state=0)

y_score_lr = lr.fit(X_train, y_train).decision_function(X_test)
```

(continues on next page)

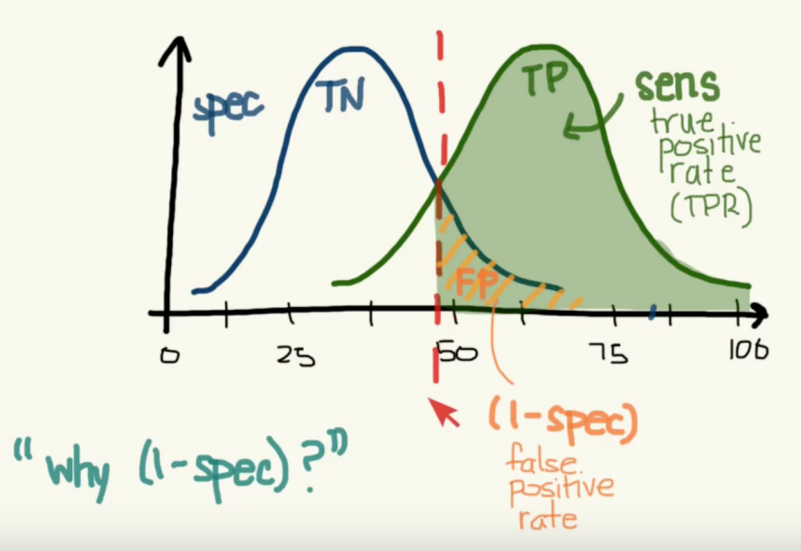


Fig. 5: Sensitivity vs 1-Specificity; or TP rate vs FP rate

(continued from previous page)

```
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_score_lr)
roc_auc_lr = auc(fpr_lr, tpr_lr)

plt.figure()
plt.xlim([-0.01, 1.00])
plt.ylim([-0.01, 1.01])
plt.plot(fpr_lr, tpr_lr, lw=3, label='LogRegr ROC curve (area = {:0.2f})'.format(roc_
↪auc_lr))
plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.title('ROC curve (1-of-10 digits classifier)', fontsize=16)
plt.legend(loc='lower right', fontsize=13)
plt.plot([0, 1], [0, 1], color='navy', lw=3, linestyle='--')
plt.axes().set_aspect('equal')
plt.show()
```

14.1.4 Log Loss

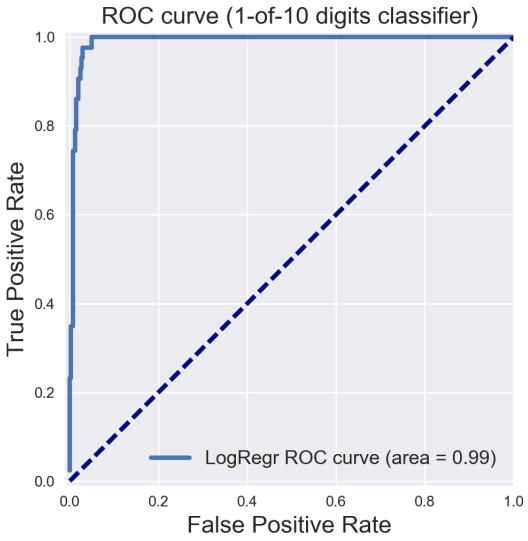
Logarithmic Loss, or Log Loss is a popular Kaggle evaluation metric, which measures the performance of a classification model where the prediction input is a probability value between 0 and 1

Log Loss quantifies the accuracy of a classifier by penalising false classifications; the catch is that Log Loss ramps up very rapidly as the predicted probability approaches 0. This article from datawookie gives a very good explanation.

14.2 Regression

For regression problems, where the response or y is a continuous value, it is common to use R-Squared and RMSE, or MAE as evaluation metrics. This website gives an excellent description on all the variants of errors metrics.

R-squared: Percentage of variability of dataset that can be explained by the model.



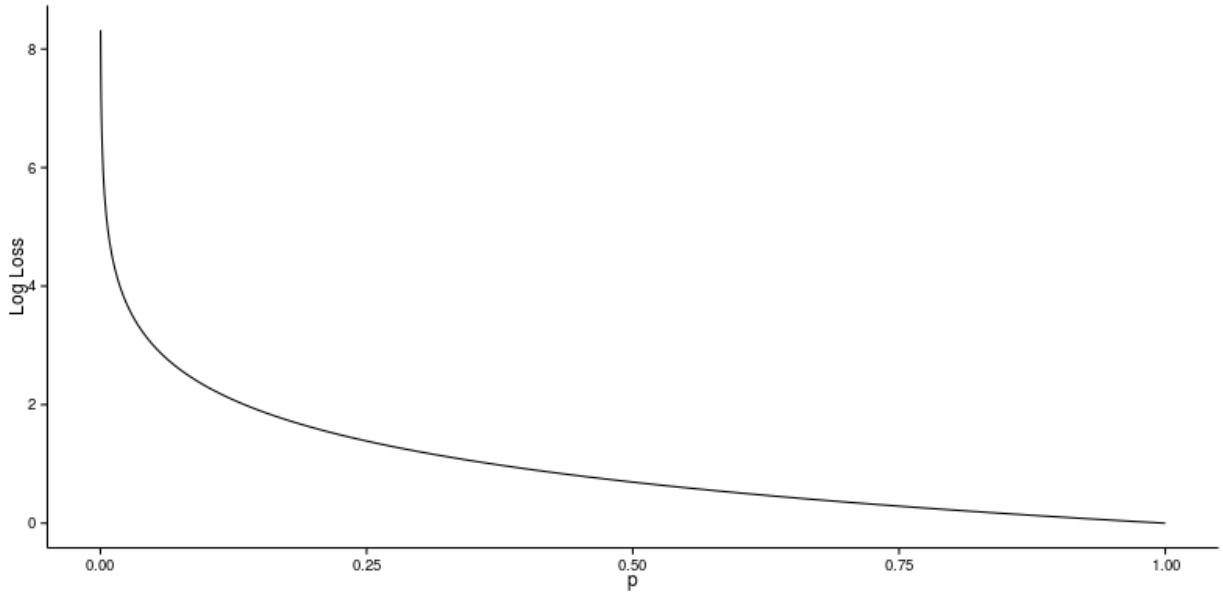


Fig. 6: From datawookie

MSE. Mean squared error. Squaring then getting the mean of all errors (so change negatives into positives).

RMSE: Squared root of MSE so that it gives back the error at the same scale (as it was initially squared).

MAE: Mean Absolute Error. For negative errors, convert them to positive and obtain all error means.

The RMSE result will always be larger or equal to the MAE. If all of the errors have the same magnitude, then RMSE=MAE. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE should be more useful when large errors are particularly undesirable.

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

forest = RandomForestRegressor(n_estimators= 375)
model3 = forest.fit(X_train, y_train)
fullmodel = forest.fit(predictor, target)
print(model3)

# R2
r2_full = fullmodel.score(predictor, target)
r2_trains = model3.score(X_train, y_train)
r2_tests = model3.score(X_test, y_test)
print('\nr2 full:', r2_full)
print('r2 train:', r2_trains)
print('r2 test:', r2_tests)

# get predictions
y_predicted_total = model3.predict(predictor)
y_predicted_train = model3.predict(X_train)
y_predicted_test = model3.predict(X_test)

# get MSE
MSE_total = mean_squared_error(target, y_predicted_total)
MSE_train = mean_squared_error(y_train, y_predicted_train)
MSE_test = mean_squared_error(y_test, y_predicted_test)

# get RMSE by squared root
print('\nTotal RMSE:', np.sqrt(MSE_total))
print('Train RMSE:', np.sqrt(MSE_train))
print('Test RMSE:', np.sqrt(MSE_test))

# get MAE
MAE_total = mean_absolute_error(target, y_predicted_total)
MAE_train = mean_absolute_error(y_train, y_predicted_train)
MAE_test = mean_absolute_error(y_test, y_predicted_test)


# Train RMSE: 11.115272389673631
# Test RMSE: 34.872611746182706

# Train MAE 8.067078668023848
# Train MAE 24.541799999999995
```

14.3 K-fold Cross-Validation

Takes more time and computation to use k-fold, but well worth the cost. By default, sklearn uses stratified k-fold cross validation. Another type is 'leave one out' cross-validation.

The mean of the final scores among each k model is the most generalised output. This output can be compared to different model results for comparison.

More here.

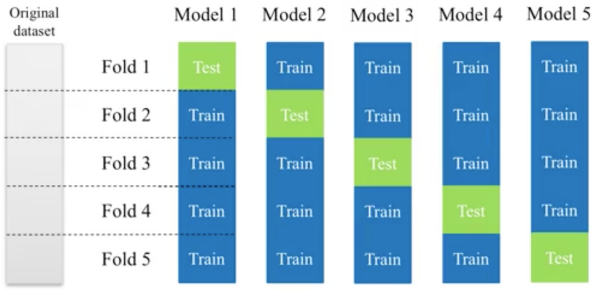


Fig. 7: k-fold cross validation, with 5-folds

```
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier

models = [('Decision Tree\t',DecisionTreeClassifier()),\
          ('Random Forest\t', RandomForestClassifier()), \
          ('KNN\t\t', KNeighborsClassifier())]

predictor = df[df.columns[1:-1]]
target = df['Cover_Type']

# using 5-fold cross validation mean scores
for clf in models:
    cv_scores = cross_val_score(clf[1], predictor, target, cv=5)
    print(clf[0], np.mean(cv_scores))

# Decision Tree         0.707473544974
# Random Forest         0.753571428571
# KNN                   0.691005291005
```

```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=2)
skf.get_n_splits(X, y)

for i in skf:
```

There are many other variants of cross validations as shown below.

14.4 Hyperparameters Tuning

There are generally 3 methods of hyperparameters tuning, i.e., Grid-Search, Random-Search, or the more automated Bayesian tuning.

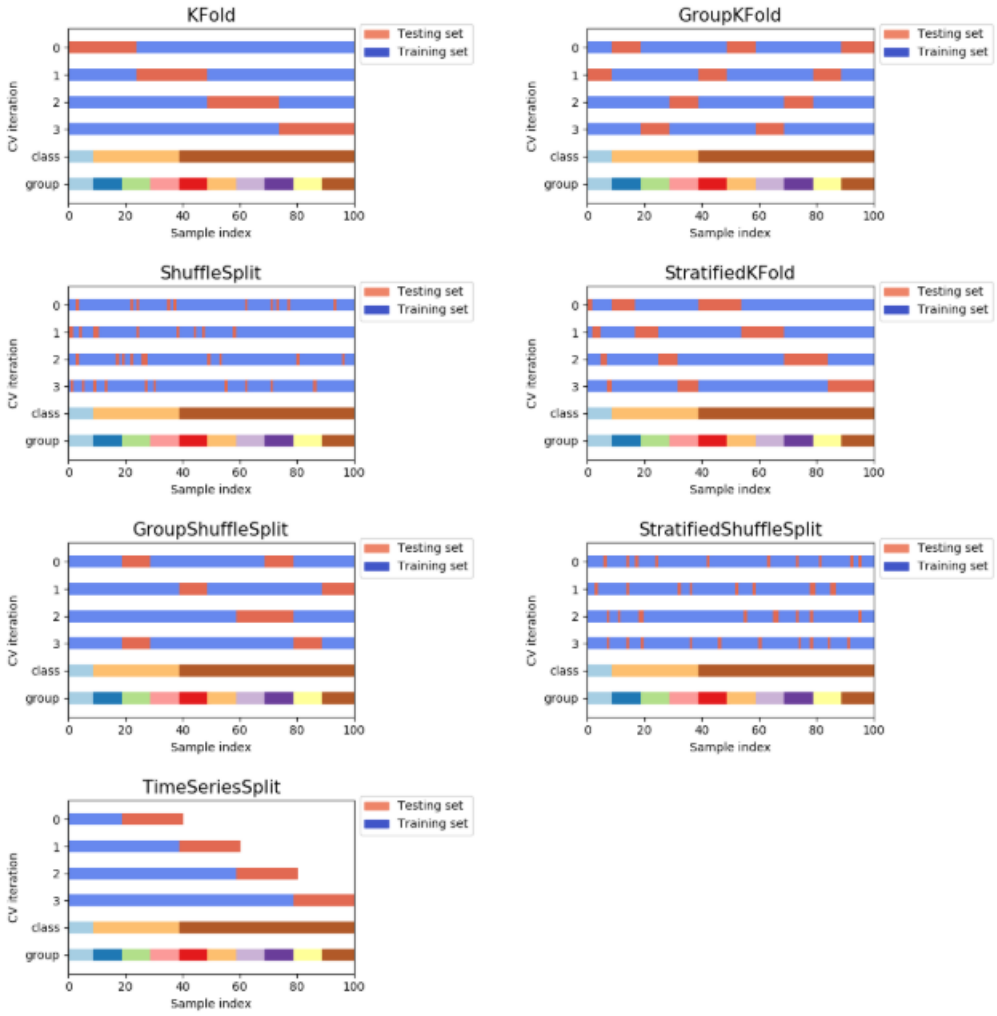


Fig. 8: Types of cross-validation available in sklearn

14.4.1 Grid-Search

From Stackoverflow: Systematically working through multiple combinations of parameter tunes, cross validate each and determine which one gives the best performance. You can work through many combination only changing parameters a bit.

Print out the `best_params_` and rebuild the model with these optimal parameters.

Simple example.

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

grid_values = {'n_estimators':[150,175,200,225]}
grid = GridSearchCV(model, param_grid = grid_values, cv=5)
grid.fit(predictor, target)

print(grid.best_params_)
print(grid.best_score_)

# {'n_estimators': 200}
# 0.786044973545
```

Others.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split


dataset = load_digits()
X, y = dataset.data, dataset.target == 1
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# choose a classifier
clf = SVC(kernel='rbf')

# input grid value range
grid_values = {'gamma': [0.001, 0.01, 0.05, 0.1, 1, 10, 100]}
# other parameters can be input in the dictionary, e.g.,
# grid_values = {'gamma': [0.01, 0.1, 1, 10], 'C': [0.01, 0.1, 1, 10]}
# OR n_estimators, max_features from RandomForest
# default metric to optimize over grid parameters: accuracy

grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, random_state=0)

grid_clf_acc.fit(X_train, y_train)
y_decision_fn_scores_acc = grid_clf_acc.decision_function(X_test)

print('Grid best parameter (max. accuracy): ', grid_clf_acc.best_params_)
print('Grid best score (accuracy): ', grid_clf_acc.best_score_)
```

Using other scoring metrics

```
# alternative metric to optimize over grid parameters: AUC
# other scoring parameters include 'recall' or 'precision'
grid_clf_auc = GridSearchCV(clf, param_grid = grid_values, scoring = 'roc_auc', cv=3,␣
↪random_state=0) # indicate AUC
grid_clf_auc.fit(X_train, y_train)
y_decision_fn_scores_auc = grid_clf_auc.decision_function(X_test)

print('Test set AUC: ', roc_auc_score(y_test, y_decision_fn_scores_auc))
print('Grid best parameter (max. AUC): ', grid_clf_auc.best_params_)
print('Grid best score (AUC): ', grid_clf_auc.best_score_)


# results 1
('Grid best parameter (max. accuracy): ', {'gamma': 0.001})
('Grid best score (accuracy): ', 0.99628804751299183)
# results 2
('Test set AUC: ', 0.99982858122393004)
('Grid best parameter (max. AUC): ', {'gamma': 0.001})
('Grid best score (AUC): ', 0.99987412783021423)


# gives break down of all permutations of gridsearch
print fittedmodel.cv_results_
# gives parameters that gives the best indicated scoring type
print CV.best_params_
```

14.4.2 Auto-Tuning

Bayesian Tuning and Bandits (BTB) is a package used for auto-tuning ML models hyperparameters. It uses Gaussian Process to do this, though there is an option for Uniform. It was born from a Master thesis by Laura Gustafson in 2018.

https://github.com/HDI-Project/BTB

```
from btb.tuning import GP
from btb import HyperParameter, ParamTypes

# remember to change INT to FLOAT where necessary
tunables = [('n_estimators', HyperParameter(ParamTypes.INT, [500, 2000])),
            ('max_depth', HyperParameter(ParamTypes.INT, [3, 20]))]

def auto_tuning(tunables, epoch, X_train, X_test, y_train, y_test, verbose=0):
    """Auto-tuner using BTB library"""
    tuner = GP(tunables)
    parameters = tuner.propose()

    score_list = []
    param_list = []

    for i in range(epoch):
        # ** unpacks dict in a argument
        model = RandomForestClassifier(**parameters, n_jobs=-1)
        model.fit(X_train, y_train)
        y_predict = model.predict(X_test)
        score = accuracy_score(y_test, y_predict)
```

(continues on next page)

(continued from previous page)

```
        # store scores & parameters
        score_list.append(score)
        param_list.append(parameters)

        if verbose==0:
            pass
        elif verbose==1:
            print('epoch: {}, accuracy: {}'.format(i+1,score))
        elif verbose==2:
            print('epoch: {}, accuracy: {}, param: {}'.format(i+1,score,parameters))

        # get new parameters
        tuner.add(parameters, score)
        parameters = tuner.propose()

    best_s = tuner._best_score
    best_score_index = score_list.index(best_s)
    best_param = param_list[best_score_index]
    print('\nbest accuracy: {}'.format(best_s))
    print('best parameters: {}'.format(best_param))
    return best_param

best_param = auto_tuning(tunables, 5, X_train, X_test, y_train, y_test)

# epoch: 1, accuracy: 0.7437106918238994
# epoch: 2, accuracy: 0.779874213836478
# epoch: 3, accuracy: 0.7940251572327044
# epoch: 4, accuracy: 0.7908805031446541
# epoch: 5, accuracy: 0.7987421383647799

# best accuracy: 0.7987421383647799
# best parameters: {'n_estimators': 1939, 'max_depth': 18}
```

Auto-Sklearn is another auto-ml package that automatically selects both the model and its hyperparameters.

https://automl.github.io/auto-sklearn/master/

```
import autosklearn.classification
import sklearn.model_selection
import sklearn.datasets
import sklearn.metrics

X, y = sklearn.datasets.load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X, y,␣
↪random_state=1)

automl = autosklearn.classification.AutoSklearnClassifier()
automl.fit(X_train, y_train)

y_pred = automl.predict(X_test)
print("Accuracy score", sklearn.metrics.accuracy_score(y_test, y_pred))
```

CHAPTER 15

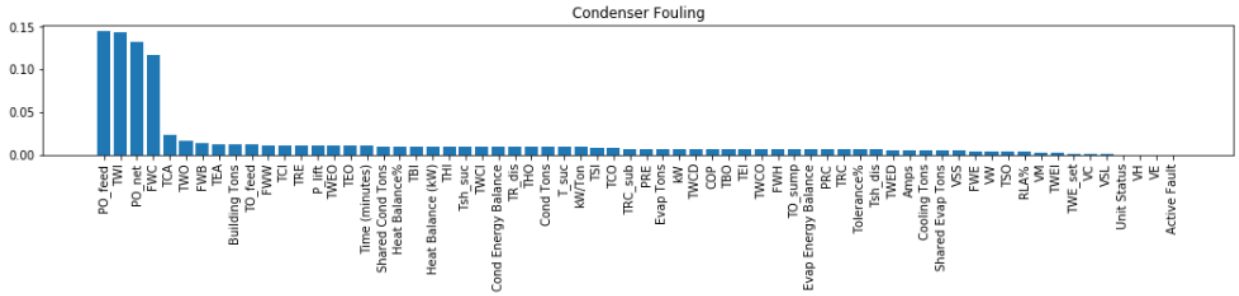
Explainability

While sklearn's supervised models are black boxes, we can derive certain plots and metrics to interprete the outcome and model better.

15.1 Feature Importance

Decision trees and other tree ensemble models, by default, allow us to obtain the importance of features.

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
%matplotlib inline

rf = RandomForestClassifier()
model = rf.fit(X_train, y_train)

# evaluation metrics
y_predict = model.predict(X_test)
accuracy = accuracy_score(y_test, y_predict)
f1 = mean(f1_score(y_test, y_predict, average=None))

# sort feature importance in df
f_impt= pd.DataFrame(model.feature_importances_, index=dataframe.columns[:-1])
f_impt = f_impt.sort_values(by=0,ascending=False)
f_impt.columns = ['feature importance']

# plot bar chart
plt.figure(figsize=(18,2))
plt.bar(f_impt.index,f_impt['feature importance'])
plt.xticks(rotation='vertical')
plt.title(fault)
```



15.2 Permutation Importance

Feature importance is a useful metric to **find the strength of each feature that contribute to a model**. However, this is only available by default in sklean tree models. This Kaggle article provides a good clear explanation of an alternative feature importance, called permutation importance, which can be used for any model. This is a third party library that needs to be installed via `pip install eli5`.

How it works is the shuffling of individual features and see how it affects model accuarcy. If a feature is important, the model accuarcy will be reduced more. If not important, the accuarcy should be affected a lot less.

Height at age 20 (cm)	Height at age 10 (cm)	...	Socks owned at age 10
182	155	...	20
175	147	...	10
...	...	...	...
156	142	...	8
153	130	...	24

Fig. 1: From Kaggle

```
import eli5
from eli5.sklearn import PermutationImportance

perm = PermutationImportance(my_model, random_state=1).fit(test_X, test_y)
eli5.show_weights(perm, feature_names = test_X.columns.tolist())
```

The output is as below. +/- refers to the randomness that shuffling resulted in. The higher the weight, the more important the feature is. Negative values are possible, but actually refer to 0; though random chance caused the predictions on shuffled data to be more accurate.

15.3 Partial Dependence Plots

While feature importance shows what variables most affect predictions, **partial dependence plots show how a feature affects predictions**. Using the fitted model to predict our outcome, and by repeatedly alter the value of just one variable, we can trace the predicted outcomes in a plot to show its dependence on the variable and when it plateaus.

Weight	Feature
0.0750 ± 0.1159	Goal Scored
0.0625 ± 0.0791	Corners
0.0437 ± 0.0500	Distance Covered (Kms)
0.0375 ± 0.0729	On-Target
0.0375 ± 0.0468	Free Kicks
0.0187 ± 0.0306	Blocked
0.0125 ± 0.0750	Pass Accuracy %
0.0125 ± 0.0500	Yellow Card
0.0063 ± 0.0468	Saves
0.0063 ± 0.0250	Offsides
0.0063 ± 0.1741	Off-Target
0.0000 ± 0.1046	Passes
0 ± 0.0000	Red
0 ± 0.0000	Yellow & Red
0 ± 0.0000	Goals in PSO
-0.0312 ± 0.0884	Fouls Committed
-0.0375 ± 0.0919	Attempts
-0.0500 ± 0.0500	Ball Possession %

Fig. 2: From Kaggle

https://www.kaggle.com/dansbecker/partial-plots

```
from matplotlib import pyplot as plt
from pdpbox import pdp, get_dataset, info_plots

# Create the data that we will plot
pdp_goals = pdp.pdp_isolate(model=tree_model, dataset=val_X,
                            model_features=feature_names, feature='Goal Scored')

# plot it
pdp.pdp_plot(pdp_goals, 'Goal Scored')
plt.show()
```

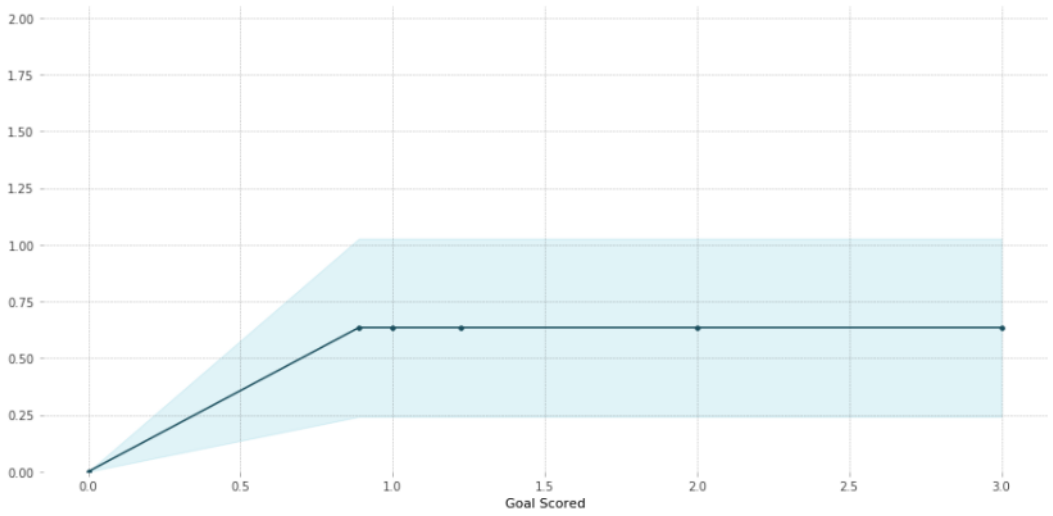


Fig. 3: From Kaggle Learn

2D Partial Dependence Plots are also useful for interactions between features.

```
# just need to change pdp_isolate to pdp_interact
features_to_plot = ['Goal Scored', 'Distance Covered (Kms)']
inter1  =  pdp.pdp_interact(model=tree_model, dataset=val_X,
                            model_features=feature_names, features=features_to_plot)

pdp.pdp_interact_plot(pdp_interact_out=inter1,
```

(continues on next page)

(continued from previous page)

```
                      feature_names=features_to_plot,
                      plot_type='contour')
plt.show()
```

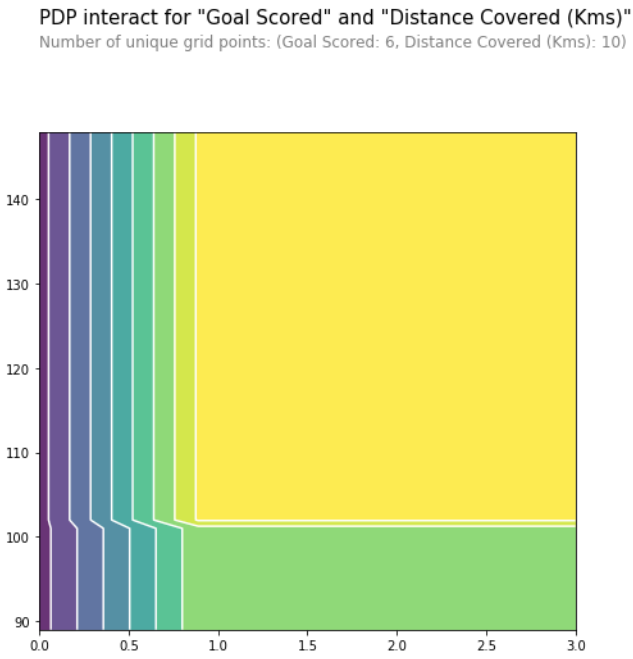


Fig. 4: From Kaggle Learn

15.4 SHAP

SHapley Additive exPlanations (SHAP) **break down a prediction to show the impact of each feature**.

https://www.kaggle.com/dansbecker/shap-values

The explainer differs with the model type:

- `shap.TreeExplainer(my_model)` for tree models
- `shap.DeepExplainer(my_model)` for neural networks
- `shap.KernelExplainer(my_model)` for all models, but slower, and gives approximate SHAP values

```
import shap  # package used to calculate Shap values

# Create object that can calculate shap values
explainer = shap.TreeExplainer(my_model)

# Calculate Shap values
shap_values = explainer.shap_values(data_for_prediction)

# load JS lib in notebook
```

(continues on next page)

(continued from previous page)

```
shap.initjs()
shap.force_plot(explainer.expected_value[1], shap_values[1], data_for_prediction)
```

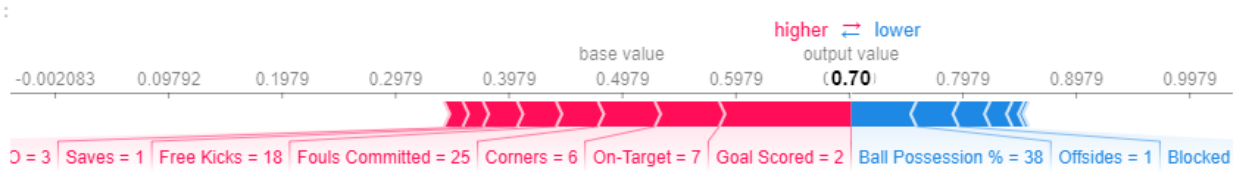


Fig. 5: From Kaggle Learn

CHAPTER 16

Docker

Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. They allow a modular construction of an application, or microservice in short. Docker is a popular tool designed to make it easier to create, deploy, and run applications by using containers.

Preprocessing scripts and models can be created as a docker image snapshot, and launched as a container in production. For models that require to be consistently updated, we need to use volume mapping such that it is not removed when the container stops running.

16.1 Creating Images

To start of a new project, create a new folder. This should only contain your docker file and related python files.

16.1.1 Dockerfile

A `Dockerfile` named as such, is a file without extension type. It contains commands to tell docker what are the steps to do to create an image. It consists of instructions & arguments.

The commands run sequentially when building the image, also known as a layered architecture. Each layer is cached, such that when any layer fails and is fixed, rebuilding it will start from the last built layer.

- `FROM` tells Docker which image you base your image on (eg, Python 3 or continuumio/miniconda3).
- `RUN` tells Docker which additional commands to execute.
- `CMD` tells Docker to execute the command when the image loads.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory inside image to /app
WORKDIR /app
```

(continues on next page)

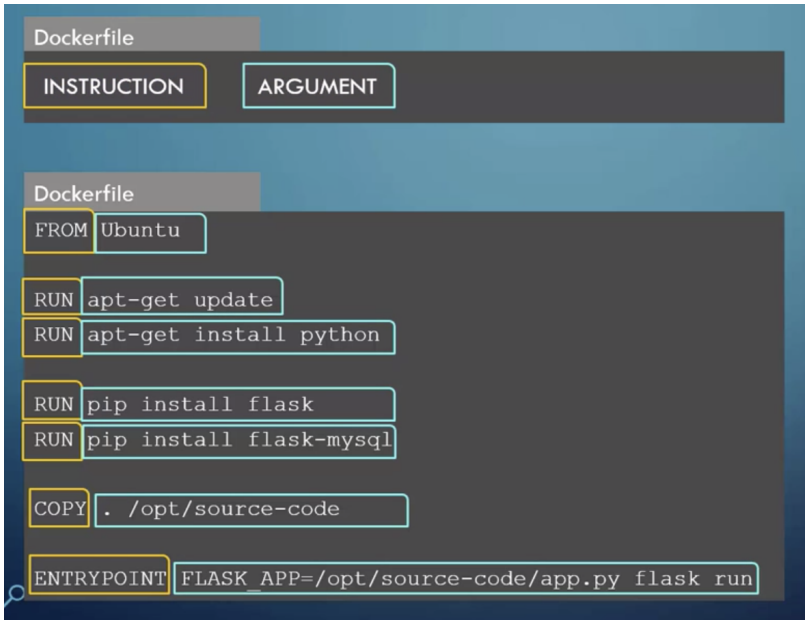


Fig. 1: from Udemy's Docker for the Absolute Beginner - Hands On

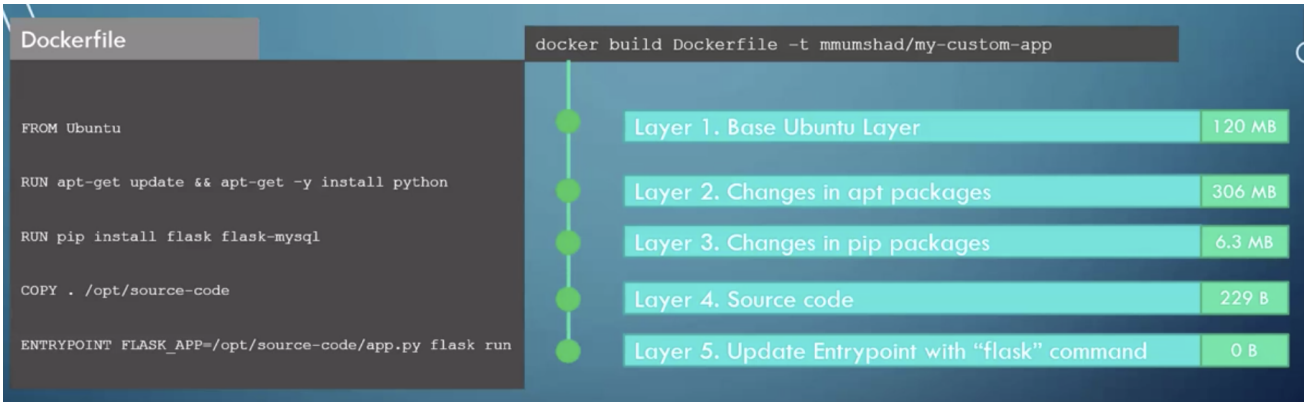


Fig. 2: from Udemy's Docker for the Absolute Beginner - Hands On

(continued from previous page)

```
# Copy the current directory contents into the image at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

16.1.2 Environment Variable

To pass environment variables from `docker run` to the python code, we can use `os.environ.get`.

```
import os
color = os.environ.get('APP_COLOR')
```

Then specify in docker run the variable for user input.

```
docker run -e APP_COLOR=green image_name
```

16.1.3 Build the Image

`docker build -t image-name .` –(-t = tag the image as) build and name image, "." as current directory to look for Dockerfile

16.1.4 Push to Dockerhub

Dockerhub is similar to Github whereby it is a repository for your images to be shared with the community. Note that Dockerhub can only allow a single image to be made private for the free account.

`docker login` –login into dockerhub, before you can push your image to the server

`docker push account/image_name` –account refers to your dockerhub account name, this tag needs to created during docker build command when building the image

16.2 Docker Compose

In a production environment, a docker compose file can be used to run all separate docker containers (which interact with each other) together. It consists of all necessary configurations that a `docker run` command provides in a yaml file.

Below is an example using wordpress blog, where both the wordpress and mysql database are needed to get it working.

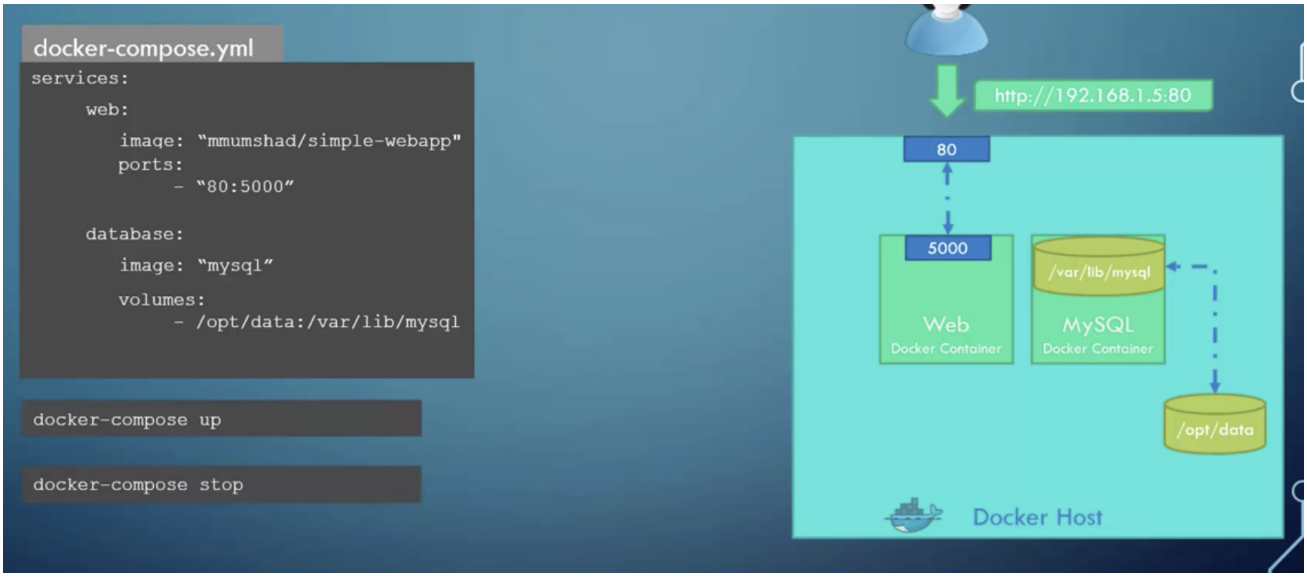


Fig. 3: from Udemy's Docker for the Absolute Beginner - Hands On

16.3 Docker Swarm

Docker Swarm allows management of multiple docker containers as clones in a cluster to ensure high availability in case of failure. This is similar to Apache Spark whereby there is a Cluster Manager (Swarm Manager), and worker nodes.

```
web:
    image: "webapp"
    deploy:
        replicas: 5
database:
    image: "mysql"
```

Use the command `docker stack deploy -c docker_compose.yml` to launch the swarm.

16.4 Networking

The **Bridge Network** is a private internal network created by Docker. All containers are attached to this network by default and they get an IP of 172.17.xxx. They are thus able to communicate with each other internally. However, to access these networks from the outside world, we need to

- map ports of these containers to the docker host.
- or associate the containers to the network host, meaning the container use the same port as the host network

If we want to separate the internal bridge networks, we can create our own internal bridge networks.

16.5 Commands

Help

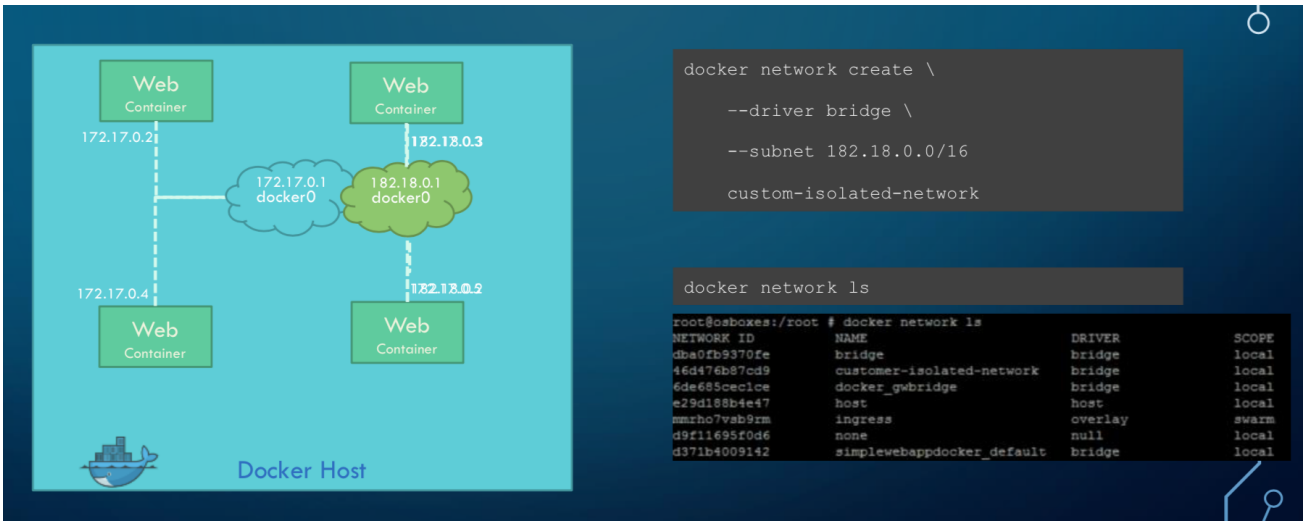


Fig. 4: from Udemy's Docker for the Absolute Beginner - Hands On

`docker --help`	list all base commands
`docker COMMAND --help`	list all options for a command

Create Image

`docker build -t image_name .`	(-t = tag the image as) build and name image, "." is the location of the dockerfile

Get Image from Docker Hub

`docker pull image_name`	pull image from dockerhub into docker
`docker run image_name COMMAND`	check if image in docker, if not pull & run image from dockerhub into docker. If no command is given, the container will stop running.
`docker run image_name cat /etc/*release*`	run image and print out the version of image

Other Run Commands

`docker run Ubuntu:17.04`	semicolon specifies the version (known as tags as listed in Dockerhub), else will pull the latest
`docker run ubuntu` vs `docker run mmumshad/ubuntu`	the first is an official image, the 2nd with the "/" is created by the community
`docker run -d image_name`	(-d = detach) docker runs in background, and you can continue typing other commands in the bash. Else need to open another terminal.
`docker run -v /local/storage/folder:/image/data/folder mysql`	(-v = volume mapping) all data will be destroyed if container is stopped

IPs & Ports

```
(base) C:\Users\Siyang>docker run -d python-barcode sleep 60
6624e63a3e6c32331565c01c970bb45d43a9b6e4f23ce6772338eb4be9974629

(base) C:\Users\Siyang>docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
6624e63a3e6c        python-barcode      "sleep 60"          6 seconds ago       Up 5 seconds                            wizardly_borg

(base) C:\Users\Siyang>
```

Fig. 5: running docker with a command. each container has a unique container ID, container name, and their base image name

`192.168.1.14`	IP address of docker host
`docker inspect container_id`	dump of container info, as well as at the bottom, under Network, the internal IP address. to view server in web browser, enter the ip and the exposed port. eg. 172.17.0.2:8080
`docker run -p 80:5000 image_name`	(host_port:container_port) map host service port with the container port on docker host

See Images & Containers in Docker

`docker images`	see all installed docker images
`docker ps`	(ps = process status) show status of images which are running
`docker ps -a`	(-a = all) show status of all images including those that had exited

Start/Stop Containers

`docker start container_name`	run container
`docker stop container_name`	stop container from running, but container still lives in the disk
`docker stop container_name1 container_name2`	stop multiple container from running in a single line
`docker stop container_id`	stop container using the ID. There is no need to type the id in full, just the first few char suffices.

Remove Containers/Images

`docker rm container_name`	remove container from docker
`docker rmi image_name`	(rmi = remove image) from docker. must remove container b4 removing image.
`docker -f rmi image_name`	(-f = force) force remove image even if container is running

Execute Commands for Containers

`docker exec container_nm COMMAND`	execute a command within container

CHAPTER 17

Ethics

17.1 Types of Errors

Accuarcy does not always tell the whole story. Think about the ramifications of different types of errors (Type I & II) from the model, and tune accordingly.

17.2 Hidden Biases

Biases based on past data will be reflected in the model. Eg., job applicant being hired could be done based on gender, race, age.

17.3 Is it Better than a Human?

Do not oversell the model capabilities. Eg. A model can predict cancer from a mammogram, but a doctor should always be there to verify the result. You never know when you need to tune the model again because of some new features that were not included in the training sample.

17.4 Model used for Unintended Purposes

The user might end up using your model for other purposes that might be unethical or wrong.

17.5 Keeping Data Confidential

While data science has the potential to help businesses and humanity in general, there are many issues in confidentially as personal data can be collected. An internet restricted environment should be conducted where such data is being extracted for analysis, or if not all sensitive data should be hashed.

Then, the question comes again, should we even collect personal data in the first place? Lets say it is mandatory to collect data from patients with mental issues, we should probably prevent collecting their identity as doing so will prevent patients from seeking help in the first place. Having high assurance of their confidentially might be the way to reduce and control their illness from escalating, and this is more important than collecting their data for analysis. After all, that is the purpose of data analysis right?

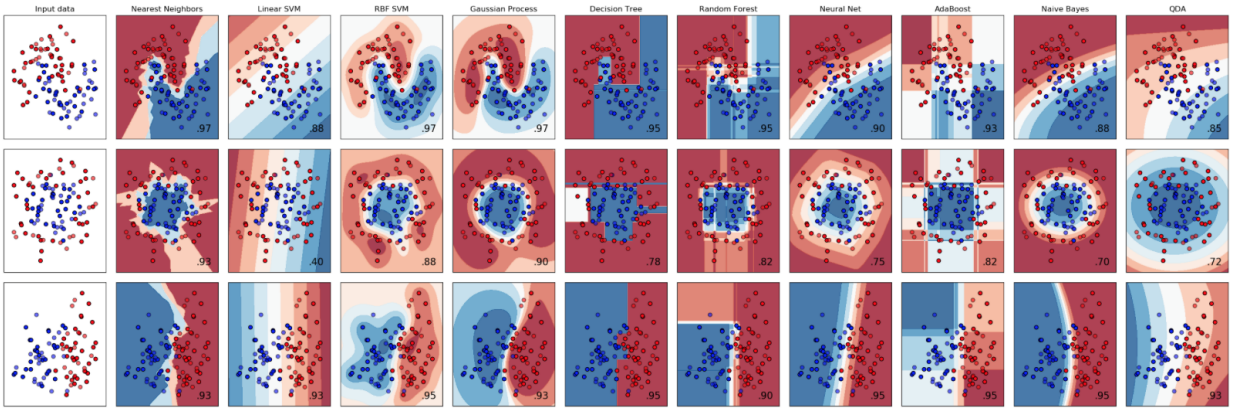
CHAPTER 18

Resources

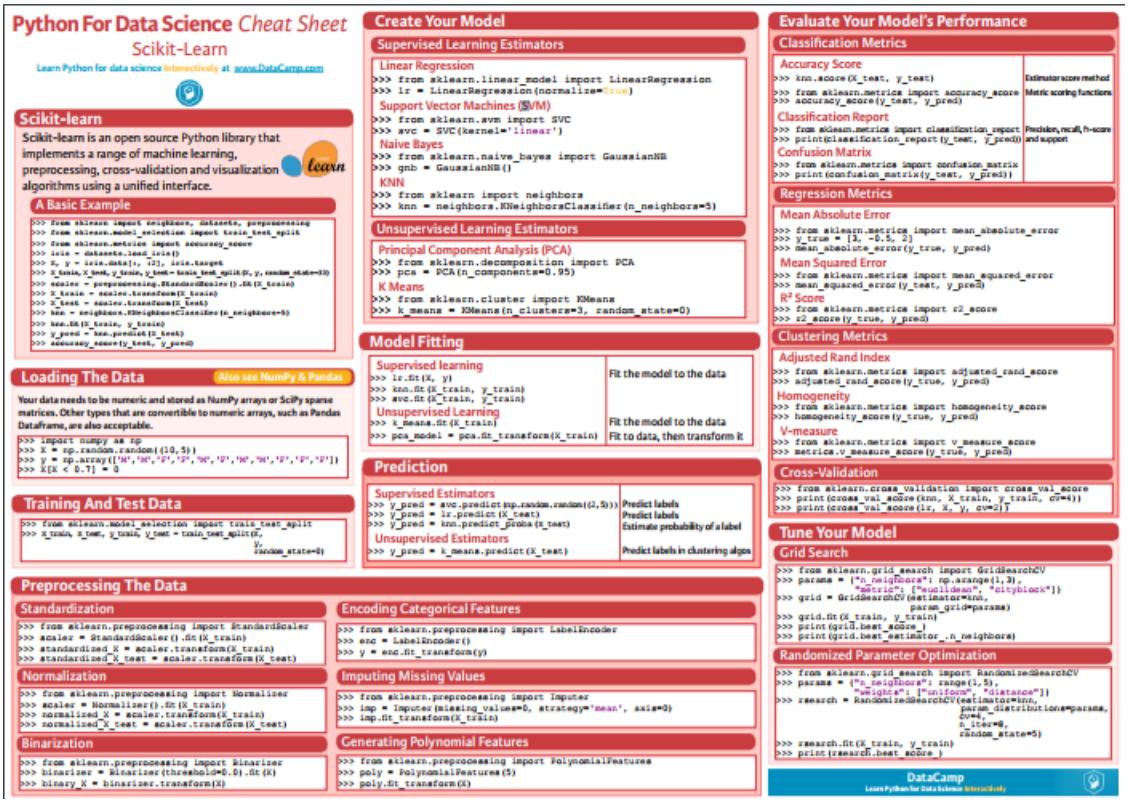
Some useful websites and video series that I refer to:

1. Setosa: Nice visual explanations of certain techniques
2. Scipy: Introduction to scipy statistical packages
3. JohnWittenauer: Blog with posts on Python machine learning
4. JakeVanderPlas: Sklearn Machine Learning guide from the man!
5. JosePortilla: Notebooks from the famous Udemy Python instructor.
6. SkLearn Tutorials: Notebooks
7. FreeCodeCamp: Very basic tutorials with videos.
8. YellowBricks: Damn simple graphing package that is compatible with sklearn!
9. Medium: 10 Essential Statistical Techniques

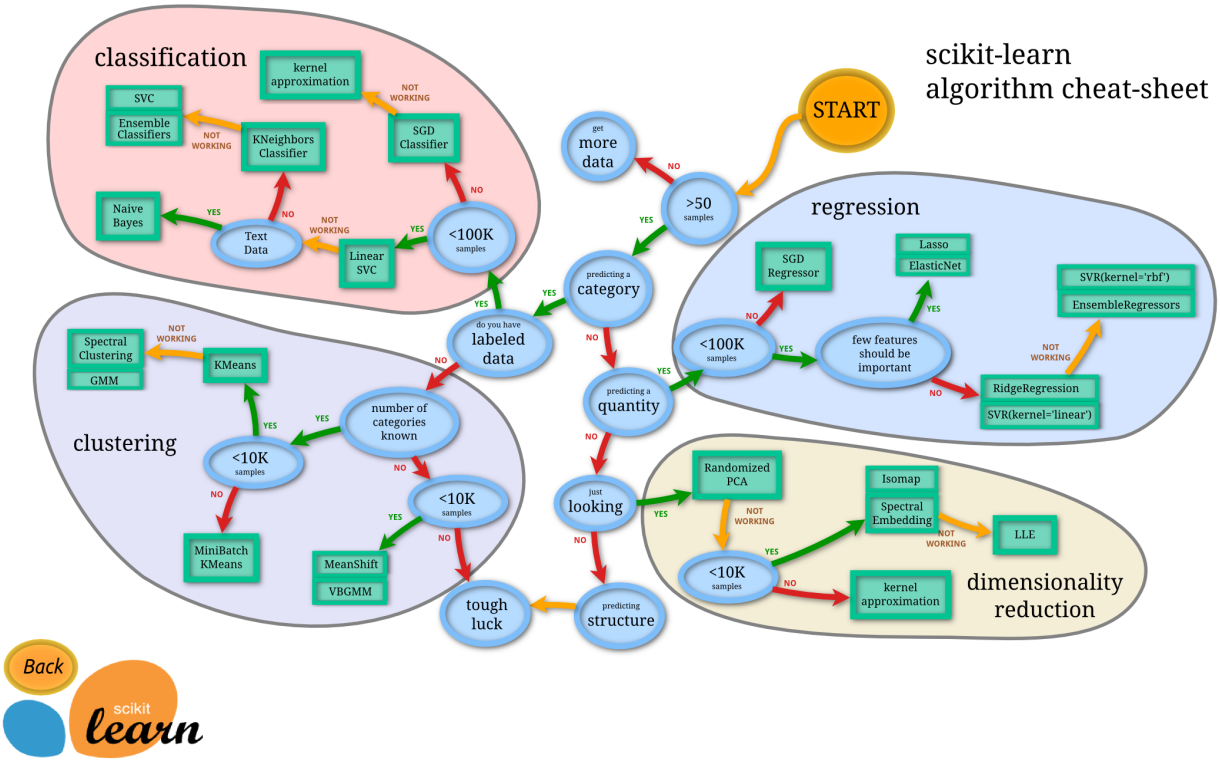
Decision Boundaries from Various Machine Learning Algorithm



Sci-Kit Learn Cheat Sheet from Data Camp



Decision Tree Map to use what Machine Learning Technique


scikit-learn
algorithm cheat-sheet
START
get more data
NO
>50 samples
YES
NO
predicting a category
YES
NO
do you have labeled data
YES
NO
predicting a quantity
YES
NO
just looking
YES
NO
predicting structure
tough luck
classification
kernel approximation
NOT WORKING
SVC
Ensemble Classifiers
NOT WORKING
KNeighbors Classifier
SGD Classifier
NO
Naive Bayes
YES
Text Data
NO
NOT WORKING
Linear SVC
YES
<100K samples
NO
regression
SGD Regressor
NO
<100K samples
YES
few features should be important
NO
Lasso
ElasticNet
YES
SVR(kernel='rbf')
EnsembleRegressors
NOT WORKING
RidgeRegression
SVR(kernel='linear')
clustering
Spectral Clustering
GMM
NOT WORKING
KMeans
YES
number of categories known
YES
NO
<10K samples
NO
MiniBatch KMeans
<10K samples
YES
NO
MeanShift
VBGMM
dimensionality reduction
Randomized PCA
NOT WORKING
YES
<10K samples
NO
Isomap
Spectral Embedding
NOT WORKING
LLE
kernel approximation
Back
scikit learn