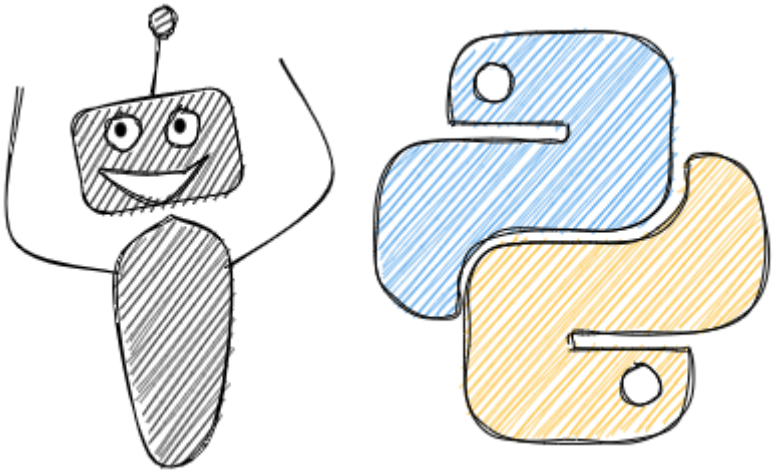
Efficient Python Tricks and Tools for Data Scientists - By Khuyen Tran

Machine Learning

This section shows some tricks and libraries for building and visualizing a machine learning model.



causalimpact: Find Causal Relation of an Event and a Variable in Python

```
!pip install pycausalimpact
```

When working with time series data, you might want to determine whether an event has an impact on some response variable or not. For example, if your company creates an advertisement, you might want to track whether the advertisement results in an increase in sales or not.

That is when causalimpact comes in handy. causalimpact analyses the differences between expected and observed time series data. With causalimpact, you can infer the expected effect of an intervention in 3 lines of code.

```
import numpy as np
import pandas as pd
from statsmodels.tsa.arima_process import
ArmaProcess
import causalimpact
from causalimpact import CausalImpact

# Generate random sample

np.random.seed(0)
```

```
ar = np.r_[1, 0.9]
ma = np.array([1])
arma_process = ArmaProcess(ar, ma)

X = 50 +
arma_process.generate_sample(nsample=1000)
y = 1.6 * X + np.random.normal(size=1000)

# There is a change starting from index 800
y[800:] += 10
```

```
data = pd.DataFrame({"y": y, "X": X}, columns=
["y", "X"])
pre_period = [0, 799]
post_period = [800, 999]
```

```
ci = CausalImpact(data, pre_period,
post_period)
print(ci.summary())
ci.plot()
```

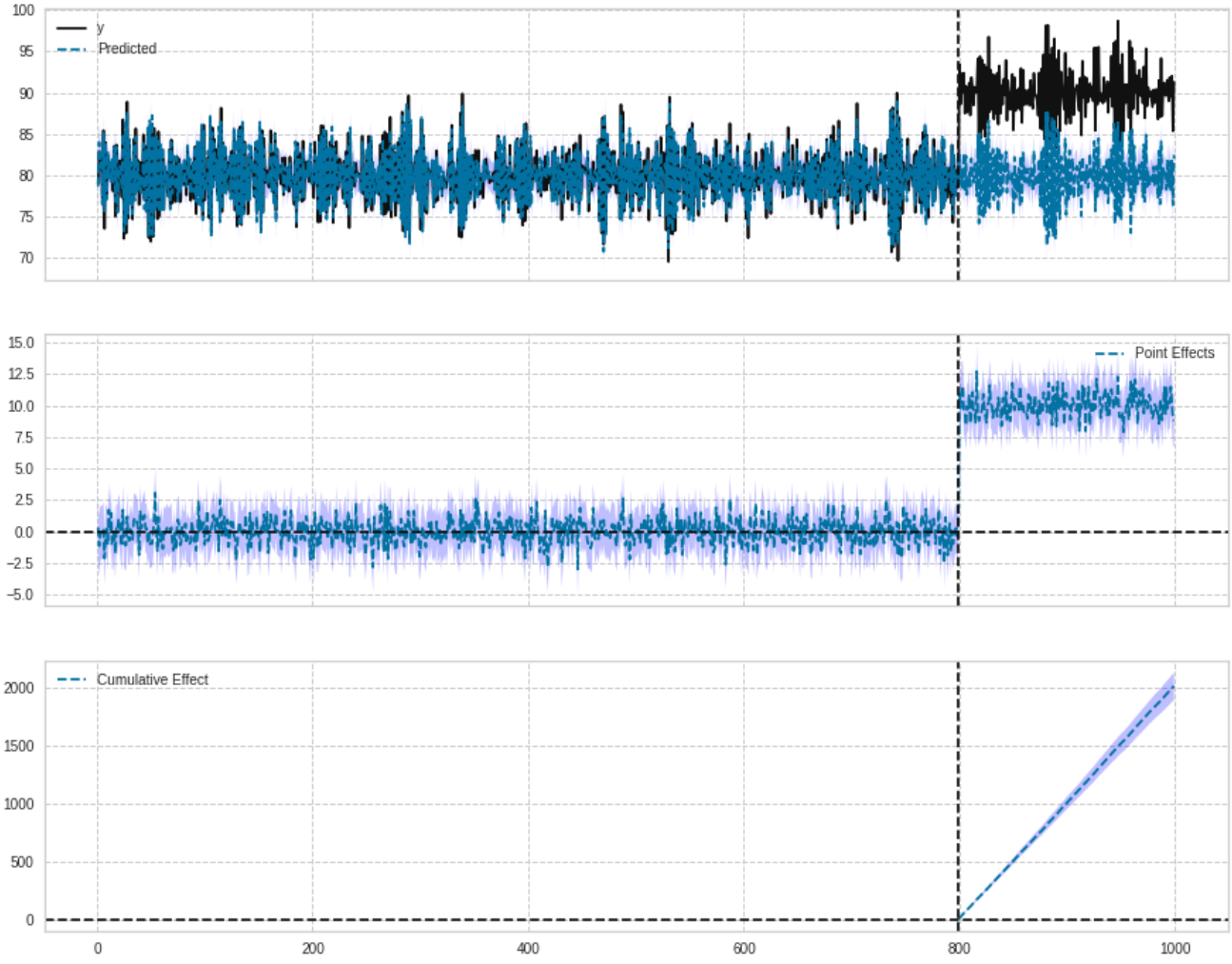
```
Posterior Inference {Causal Impact}
                          Average
Cumulative
Actual                    90.03
18006.16
Prediction (s.d.)         79.97 (0.3)
15994.43 (60.49)
95% CI                    [79.39, 80.58]
[15878.12, 16115.23]
```

```
Absolute effect (s.d.)    10.06 (0.3)
2011.72 (60.49)
95% CI                    [9.45, 10.64]
[1890.93, 2128.03]

Relative effect (s.d.)    12.58% (0.38%)
12.58% (0.38%)
95% CI                    [11.82%, 13.3%]
[11.82%, 13.3%]

Posterior tail-area probability p: 0.0
Posterior prob. of a causal effect: 100.0%

For more details run the command:
print(impact.summary('report'))
Analysis report {CausalImpact}
```



Note: The first 1 observations were removed due to approximate diffuse initialization.

Pipeline + GridSearchCV: Prevent Data Leakage when Scaling the Data

Scaling the data before using GridSearchCV can lead to data leakage since the scaling tells some information about the entire data. To prevent this, assemble both the scaler and machine learning models in a pipeline then use it as the estimator for GridSearchCV.

```
from sklearn.model_selection import
train_test_split, GridSearchCV
from sklearn.preprocessing import
StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC
from sklearn.datasets import load_iris

# load data
df = load_iris()
X = df.data
y = df.target

# split data into train and test
X_train, X_test, y_train, y_test =
train_test_split(X, y, random_state=0)
```

```
# Create a pipeline variable
make_pipe = make_pipeline(StandardScaler(),
SVC())

# Defining parameters grid
grid_params = {"svc__C": [0.1, 1, 10, 100,
1000], "svc__gamma": [0.1, 1, 10, 100]}

# hypertuning
grid = GridSearchCV(make_pipe, grid_params,
cv=5)
grid.fit(X_train, y_train)

# predict
y_pred = grid.predict(X_test)
```

The estimator is now the entire pipeline instead of just the machine learning model.

squared=False: Get RMSE from Sklearn's mean_squared_error method

If you want to get the root mean squared error using sklearn, pass `squared=False` to sklearn's `mean_squared_error` method.

```
from sklearn.metrics import mean_squared_error

y_actual = [1, 2, 3]
y_predicted = [1.5, 2.5, 3.5]
rmse = mean_squared_error(y_actual,
y_predicted, squared=False)
rmse
```

```
0.5
```

modelkit: Build Production ML Systems in Python

```
!pip install modelkit textblob
```

If you want your ML models to be fast, type-safe, testable, and fast to deploy to production, try modelkit. modelkit allows you to incorporate all of these features into your model in several lines of code.

```
from modelkit import ModelLibrary, Model
from textblob import TextBlob, WordList
# import nltk
# nltk.download('brown')
# nltk.download('punkt')
```

To define a modelkit Model, you need to:

- create class inheriting from `modelkit.Model`
- implement a `_predict` method

```
class NounPhraseExtractor(Model):

    # Give model a name
    CONFIGURATIONS = {"noun_phrase_extractor":
{}}

    def _predict(self, text):
        blob = TextBlob(text)
        return blob.noun_phrases
```

You can now instantiate and use the model:

```
noun_extractor = NounPhraseExtractor()
noun_extractor("What are your learning
strategies?")
```

```
WordList(['learning strategies'])
```

You can also create test cases for your model and make sure all test cases are passed.

```
class NounPhraseExtractor(Model):

    # Give model a name
    CONFIGURATIONS = {"noun_phrase_extractor":
{}}

    TEST_CASES = [
        {"item": "There is a red apple on the
tree", "result": WordList(["red apple"])}
    ]

    def _predict(self, text):
        blob = TextBlob(text)
        return blob.noun_phrases
```

```
noun_extractor = NounPhraseExtractor()
noun_extractor.test()
```

```
TEST 1: SUCCESS
```

modelkit also allows you to organize a group of models using `ModelLibrary`.

```
class SentimentAnalyzer(Model):

    # Give model a name
    CONFIGURATIONS = {"sentiment_analyzer":
{}}

    def _predict(self, text):
        blob = TextBlob(text)
        return blob.sentiment
```

```
nlp_models = ModelLibrary(models=
[NounPhraseExtractor, SentimentAnalyzer])
```

Get and use the models from `nlp_models`.

```
noun_extractor =
model_collections.get("noun_phrase_extractor")
noun_extractor("What are your learning
strategies?")
```

```
WordList(['learning strategies'])
```

```
sentiment_analyzer =
model_collections.get("sentiment_analyzer")
sentiment_analyzer("Today is a beautiful
day!")
```

```
Sentiment(polarity=1.0, subjectivity=1.0)
```

Link to modelkit.

Decompose high dimensional data into two or three dimensions

```
!pip install yellowbrick
```

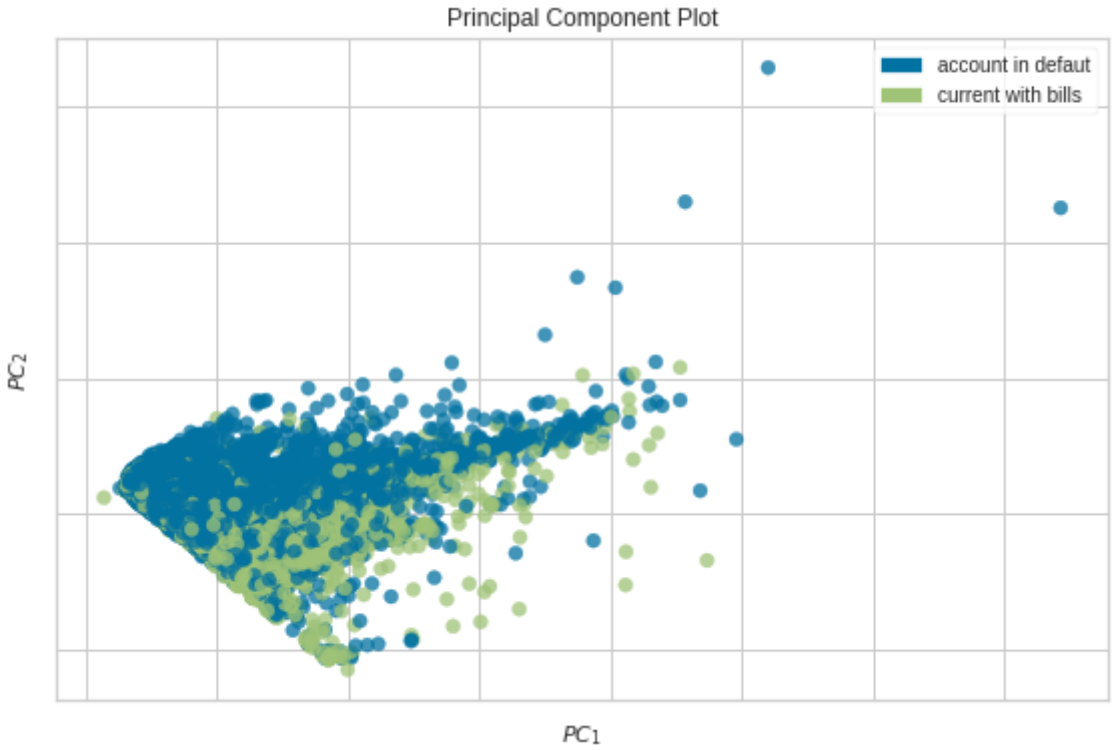
If you want to decompose high dimensional data into two or three dimensions to visualize it, what should you do? A common technique is PCA. Even though PCA is useful, it can be complicated to create a PCA plot.

Lucikily, Yellowbrick allows you visualize PCA in a few lines of code

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import PCA
```

```
X, y = load_credit()
classes = ["account in defaut", "current with bills"]
```

```
visualizer = PCA(scale=True, classes=classes)
visualizer.fit_transform(X, y)
visualizer.show()
```



Link to Yellowbrick.

Visualize Feature Importances with Yellowbrick

Having more features is not always equivalent to a better model. The more features a model has, the more sensitive the model is to errors due to variance. Thus, we want to select the minimum required features to produce a valid model.

A common approach to eliminate features is to eliminate the ones that are the least important to the model. Then we re-evaluate if the model actually performs better during cross-validation.

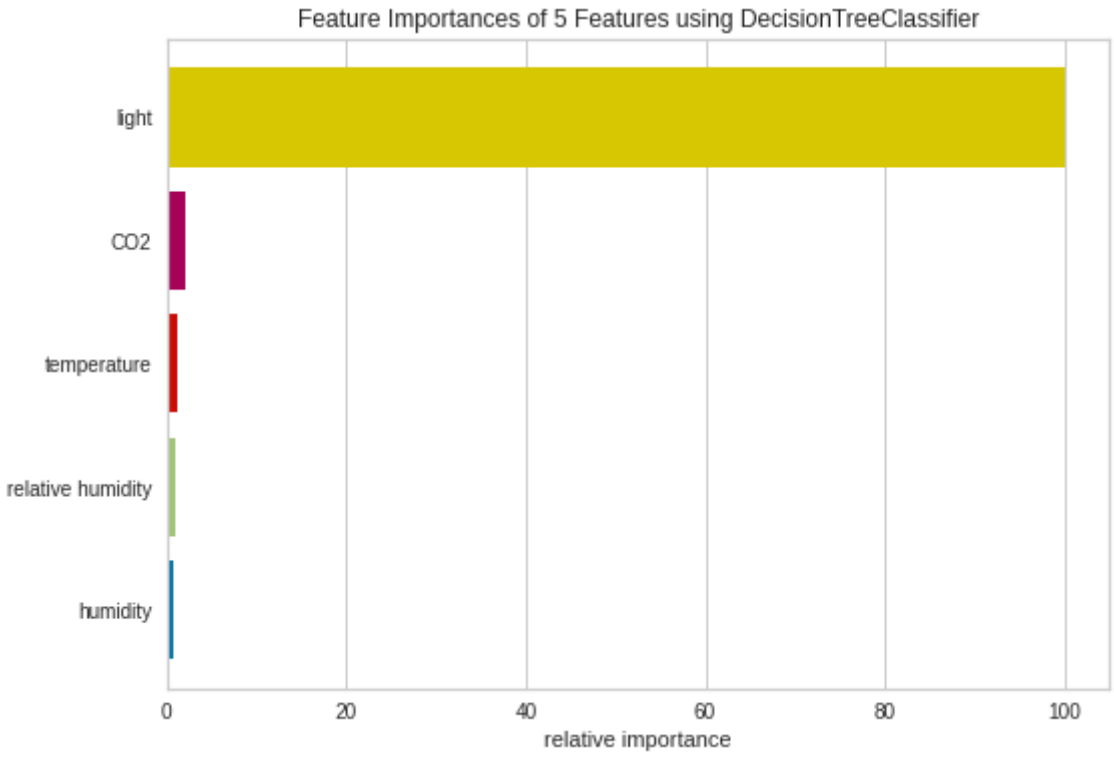
Yellowbrick's `FeatureImportances` is ideal for this task since it helps us to visualize the relative importance of the features for the model.

```
from sklearn.tree import
DecisionTreeClassifier
from yellowbrick.datasets import
load_occupancy
from yellowbrick.model_selection import
FeatureImportances
```

```
X, y = load_occupancy()

model = DecisionTreeClassifier()

viz = FeatureImportances(model)
viz.fit(X, y)
viz.show();
```



From the plot above, it seems like the light is the most important feature to DecisionTreeClassifier, followed by CO2, temperature.

Link to Yellowbrick.

My full article about Yellowbrick.

Mlxtend: Plot Decision Regions of Your ML Classifiers

```
!pip install mlxtend
```

How does your machine learning classifier decide which class a sample belongs to? Plotting a decision region can give you some insights into your ML classifier's decision.

An easy way to plot decision regions is to use mlxtend's `plot_decision_regions`.

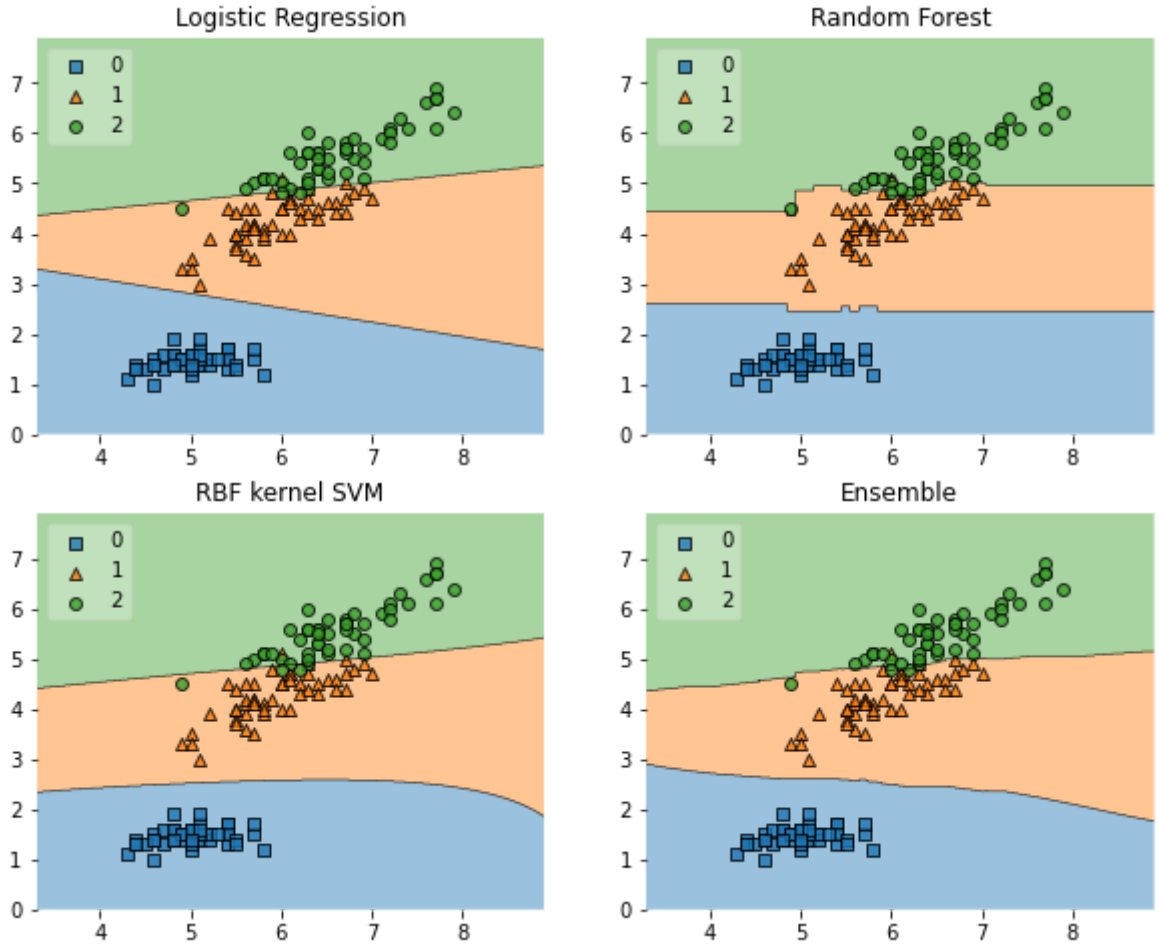
```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import itertools
from sklearn.linear_model import
LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import
RandomForestClassifier
from mlxtend.classifier import
EnsembleVoteClassifier
from mlxtend.data import iris_data
from mlxtend.plotting import
plot_decision_regions
```

```
# Initializing Classifiers
clf1 = LogisticRegression(random_state=0)
clf2 = RandomForestClassifier(random_state=0)
clf3 = SVC(random_state=0, probability=True)
eclf = EnsembleVoteClassifier(clfs=[clf1,
clf2, clf3], weights=[2, 1, 1], voting='soft')

# Loading some example data
X, y = iris_data()
X = X[:,[0, 2]]

# Plotting Decision Regions
gs = gridspec.GridSpec(2, 2)
fig = plt.figure(figsize=(10, 8))

for clf, lab, grd in zip([clf1, clf2, clf3,
eclf],
                         ['Logistic
Regression', 'Random Forest', 'RBF kernel
SVM', 'Ensemble'],
                         itertools.product([0,
1], repeat=2)):
    clf.fit(X, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y,
clf=clf, legend=2)
    plt.title(lab)
plt.show()
```



Mlxtend (machine learning extensions) is a Python library of useful tools for the day-to-day data science tasks.

Find other useful functionalities of Mlxtend here.