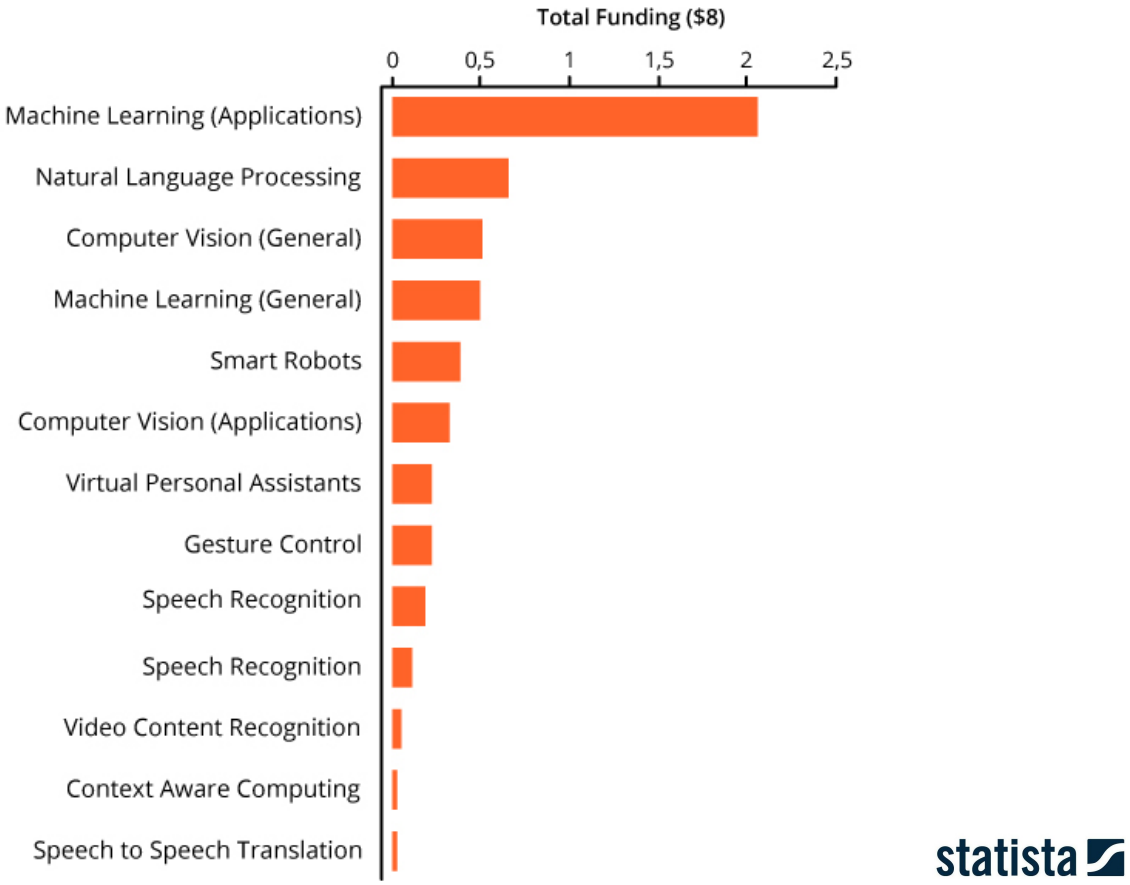
Whitepaper

What to expect while designing a Machine Learning solution?

ITSVIT

What to expect while designing a Machine Learning solution?

Choosing the correct Machine Learning (ML) model can be hugely beneficial for various business processes and dealing with specific tasks that cannot be solved the other way.



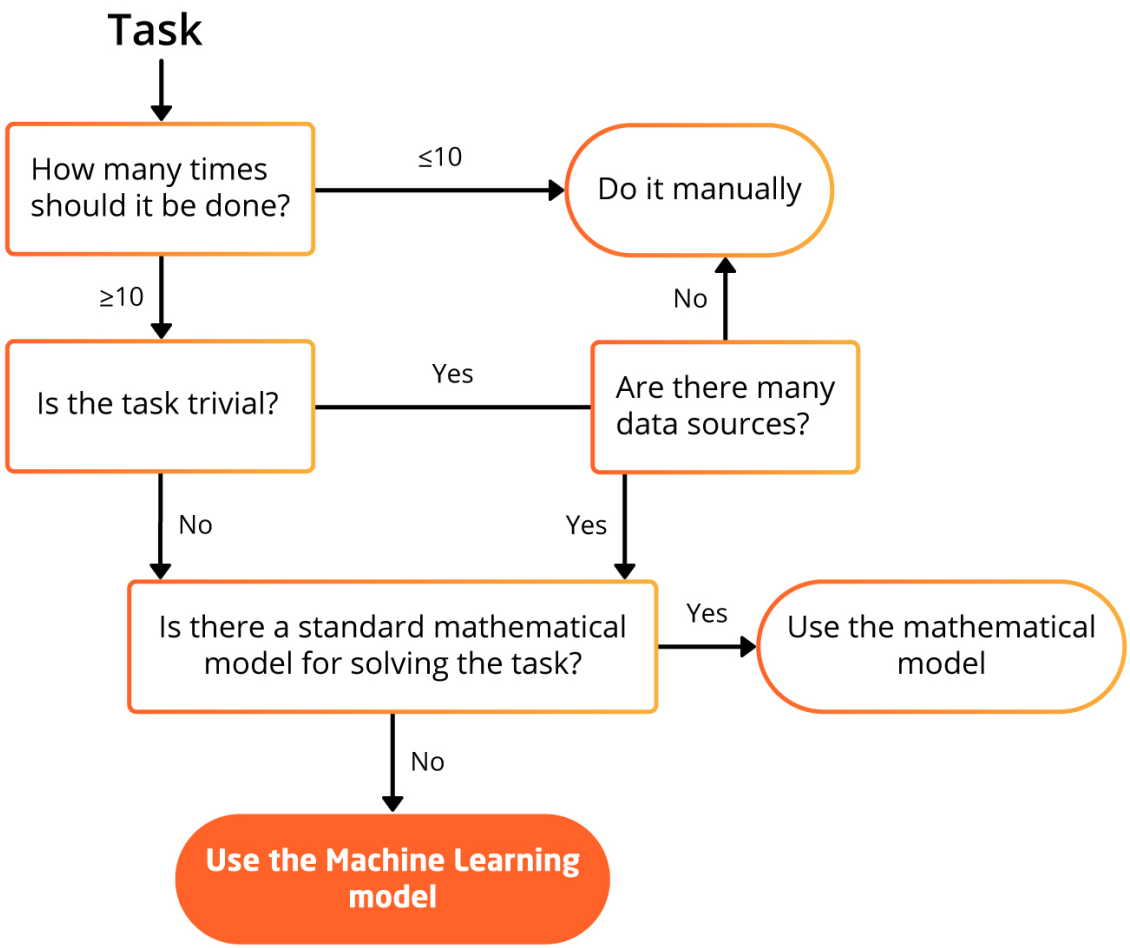
However, there is no one-size-fits-all Machine Learning solution for every case. Thus said, choosing the correct model and training it might require a significant time and resources. This is why knowing what to expect while designing a Machine Learning solution is crucial for coming up with realistic budget and estimates and overall success of the project.

1 Step

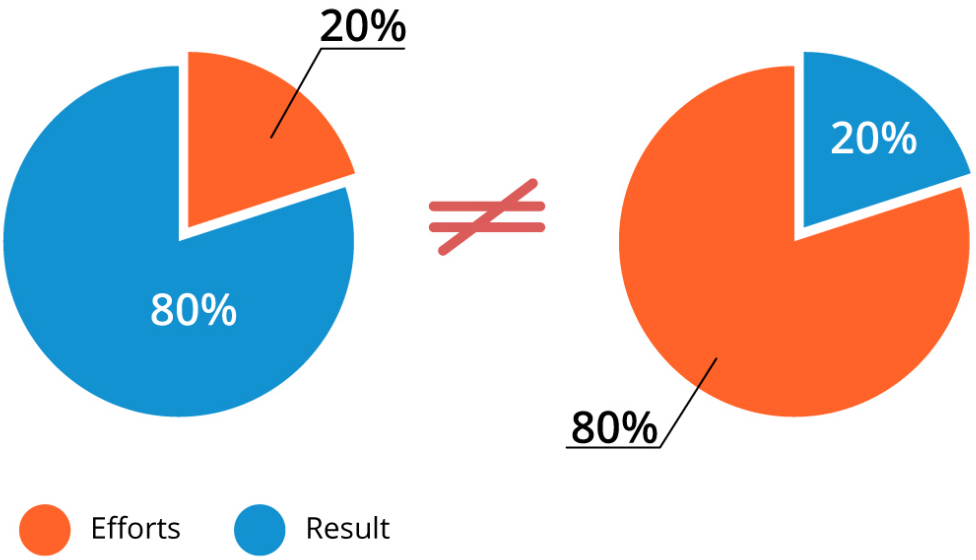
Task analysis

Once a customer wants us to build an ML solution for him, we have to answer three major questions:

1 **Does this solution have to include an ML algorithm?**
To be frank, 2/3 of cases can be solved using simple methods of mathematical logic. To say even more, many typical data analysis tasks have well-documented solutions and we can relatively easy apply or adjust one of those for any case at hand.



2 **Will we be able to guarantee sufficient quality of the resulting data?**
In data science and ML field, the following works as a rule of thumb: ***applying 20% of effort ensures reaching 80% of accuracy, while the remaining 80% of effort don't ensure reaching the remaining 20% of quality.*** As none ML model can be absolutely accurate we have to set a threshold of feasible application of our time and effort.



3 **What ML model will fit the case?**
There is a huge variety of ML models applicable to multiple business cases. However, the models best suited to visual image processing will be of little use when dealing with financial data, obviously. We have to select 3-4 most appropriate models

Here is the approximate process of selecting the ML algorithm that will fit the requirements best.

Let's assume our task is to find the most accurate model for prediction of some value dispersion over the course of several years, able to produce the results fast.

We have the historical data for 20 years, from 1980-01-02 to 1999-12-31. We need to launch the Matplotlib to make it usable by the ML models:

```
In [1]: %matplotlib inline
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
```

We should represent it as a data frame with dots as a decimal separator:

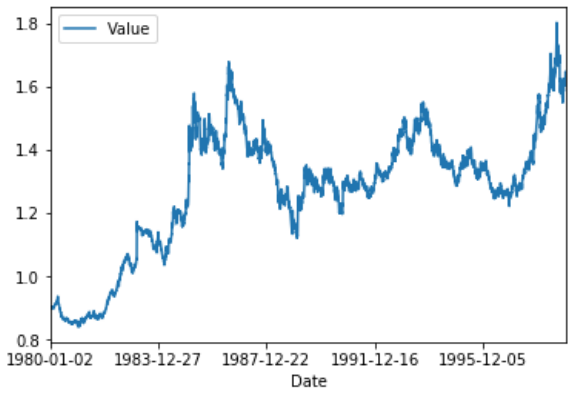
```
In [2]: df = pd.read_csv('time_series.csv', header=None, decimal=",")
df.columns = ['Date', 'Value' ]
```

We must make sure the duplication and all values (dates) are unique:

```
In [3]: len(set(df['Date'])) == len(df)
Out[3]: True
```

Now we can build a plot graph to represent a value's dispersion over the years.

```
In [4]: df.plot(x='Date', y='Value')
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5116444d10>
```



As we uncovered while working with the data set, it has some empty spaces within it (it contains 4.800 lines instead of 7.300, which is 365x20). The data before and after the skipped dates did not change much, so we decided to fill the gaps with the last values specified:

In [5]:
```
ts = df.set_index('Date')

all_dates = pd.date_range('1980-01-02', '1996-12-31')
ts.index = pd.DatetimeIndex(ts.index)
ts = ts.reindex(all_dates)
ts.fillna(method='bfill', inplace=True)
ts.head()
```

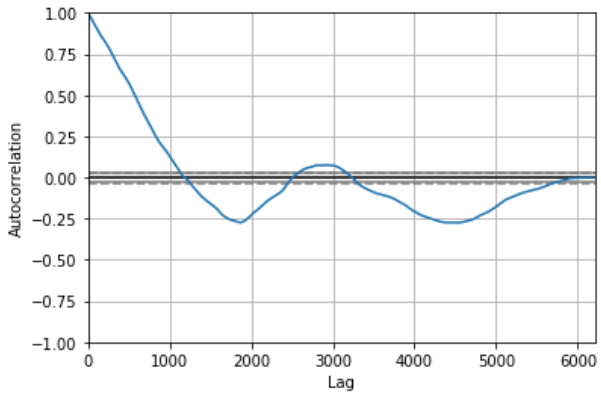
Out[5]:

	Value
1980-01-02	0.903
1980-01-03	0.901
1980-01-04	0.902
1980-01-05	0.898
1980-01-06	0.898

The autocorrelation graph confirms our assumptions that the value changes smoothly, without drastic spikes.

In [6]:
```
pd.plotting.autocorrelation_plot(ts)
```

Out[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5115ba7a10>
```

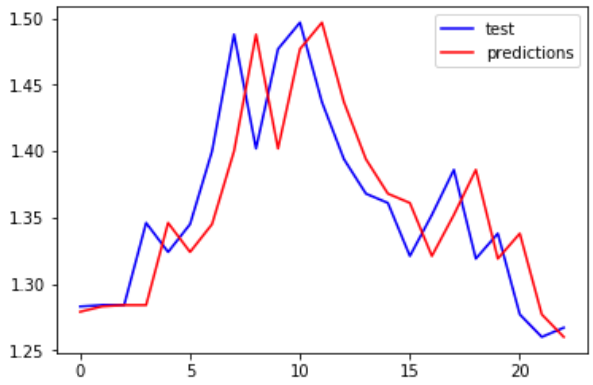


Thus said, we can limit our train to a certain excerpt of the data set. Let's work with every 90th value and use the Mean Squared Error metrics.

In [7]:
```
X = ts.values[::90]
train_size = int(len(X) * 0.66)
train, test = X[:train_size], X[train_size:]


def analyze_predictions(test_values, predictions):
    score = mean_squared_error(test_values, predictions)
    print('Score: %.3f MSE' % (score))
    plt.plot(test_values, 'b', label='test')
    plt.plot(predictions, 'r', label='predictions')
    plt.legend()
```

We will take the previous value as the baseline.

```
In [8]: analyze_predictions(test[1:], test[:-1])
Score: 0.002 MSE
```



The best models to use for the task appear to be ARIMA, LSTM and the Random forest.
The full code gist for this stage is available on Github.

2 Step

Solution implementation and ML model training

Once we select 3-4 most appropriate ML models for the case we begin training them to choose the optimal one. In order to provide the most precise results, the training should take a long time and include processing large data sets. We will make some tweaks and adjustments during the process to ensure maximum precision of the training outcomes. This means the trained ML model should provide the results that match or are very similar to the actual historical data.

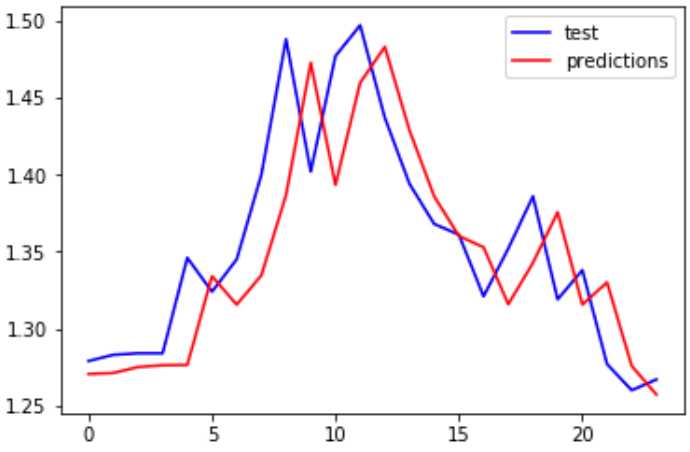
Should the customer be able to provide training datasets at once — we begin training the model. In case the training data should be procured beforehand — we build some scrapers and form the training data set prior to training the model, the process described in more details in our 4-step guide to building a Big Data solution

ARIMA model

The first model we tried to apply was ARIMA. As the basic AR, I, MA parameters have little impact on the prediction accuracy, we will use the least complicated model.

In [9]:
```
from statsmodels.tsa.arima_model import ARIMA


def get_arima_predictions(train, test, ORDER=(2, 0, 0)):
    history = [x for x in train]
    predictions = list()

    for test_index, test_value in enumerate(test):
        model = ARIMA(history, ORDER)
        model_fit = model.fit(disp=0)
        output = model_fit.forecast()
        prediction = output[0]
        predictions.append(prediction)
        history.append(test_value)
    return predictions


predictions = get_arima_predictions(train, test)
analyze_predictions(test, predictions)
```

```
Score: 0.002 MSE
```

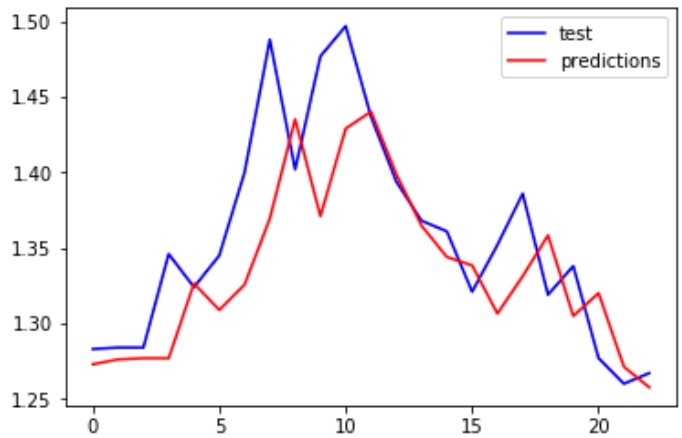


LSTM model

The next model we tried to apply was LSTM. The key parameter affecting the extent of error here was the number of epochs, though its impact decreases every 100 epochs if we set the “verbose=2”.

In [10]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM


def create_dataset(series):
    # we'll use the previous value to predict the next one
    return series[:-1], series[1:]


trainX, trainY = create_dataset(train)
testX, testY = create_dataset(test)

# reshape input into [samples, time steps=1, features=1]
trainX = np.reshape(trainX, (trainX.shape[0], 1, 1))
testX = np.reshape(testX, (testX.shape[0], 1, 1))

# create LSTM network
model = Sequential()
model.add(LSTM(2, input_shape=(1, 1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

# train LSTM netrwork
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=0)


predictions = model.predict(testX)
analyze_predictions(testY, predictions)
```

```
Score: 0.002 MSE
```

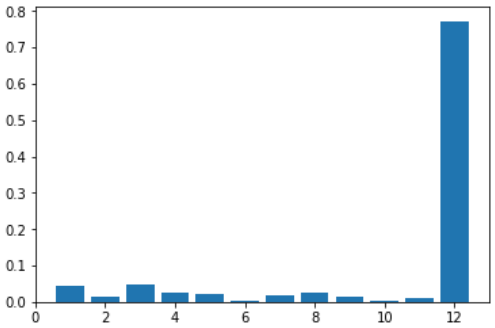


As you can see, LSTM model was quite accurate, yet not so efficient as compared to the ARIMA model.

Random Decision Forest model

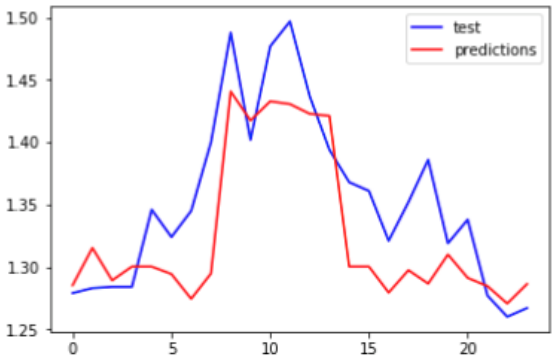
We used the last 12 observations as the input for this model to provide a sufficient excerpt. The last value was obviously the most influential for the prediction.

```
In [11]: from sklearn.ensemble import RandomForestRegressor

         shift = 12
         reshaped_X = np.array([X[i : i + shift + 1] for i in range(0, len(X) - shift, 1)])
         reshaped_train_size = int(len(X) * 0.66) - shift
         reshaped_train, reshaped_test = reshaped_X[:reshaped_train_size, :, 0],
           reshaped_X[reshaped_train_size:, :, 0]

         clf = RandomForestRegressor(n_estimators=10)
         clf.fit(reshaped_train[:, :shift], reshaped_train[:, shift])
         plt.bar(range(1, 1 + shift), clf.feature_importances_)

Out[11]: <Container object of 12 artists>
```



Using the trained model versus the existing dataset:

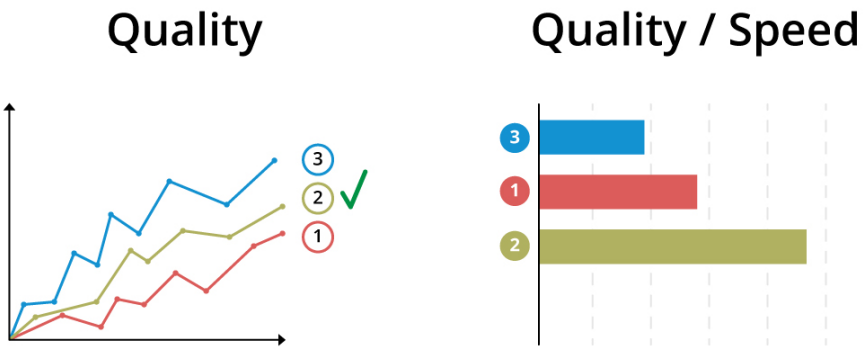
```
In [12]: predictions = clf.predict(reshaped_test[:, :shift])
         test_values = reshaped_test[:, shift]

         analyze_predictions(test_values, predictions)

         Score: 0.002 MSE
```



The full code gist for this step is available on Github.

As you can see, applying the ARIMA model was the best for this case, both in terms of accuracy of predictions and the speed of data processing.

Once we complete training the ML algorithms, we build the graphs illustrating the results of the data processing by each of the models. These generally highlight 2 major concerns:



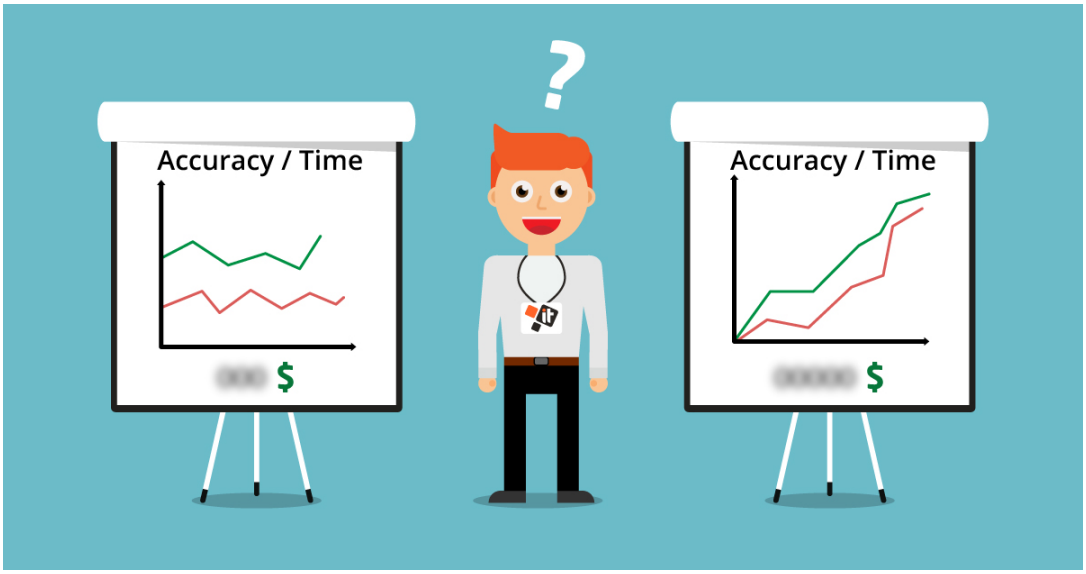
3 Step

Fine-tuning the solution

By the moment we deliver a bunch of trained ML models that will get the job done (with varying degree of efficiency), the customers usually provide some additional input. They either want to add more data to the pool (like combining the flows from several companies) or add more parameters, or increase the speed of data processing, or decrease the costs.

To ensure our models fit the updated requirements we fine-tune them, either by applying additional requirements or developing bespoke solutions fitting the case. We make sure the chosen solution will suffice the customer's requirements for 3 years at least.

Sometimes it is obvious the most accurate solution is not the fastest one. In that case, we conduct a meeting with a customer and select the optimal strategy — precise but slower, or more productive but somewhat less accurate.



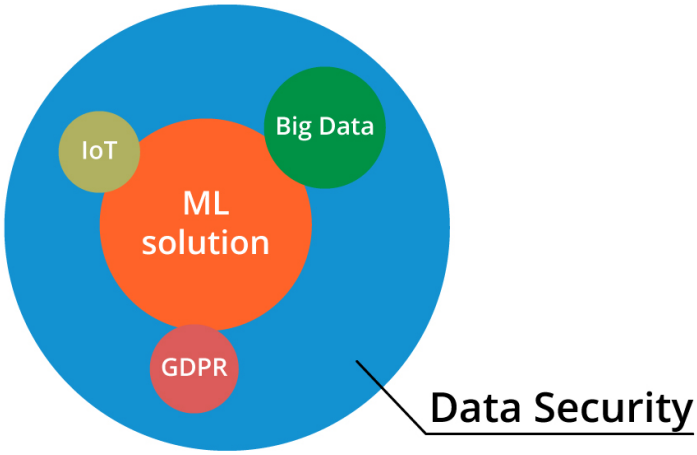
We can also deploy several ML algorithms in parallel to provide the best of both worlds: high speed for daily analysis and high precision for special projects.

4 Step

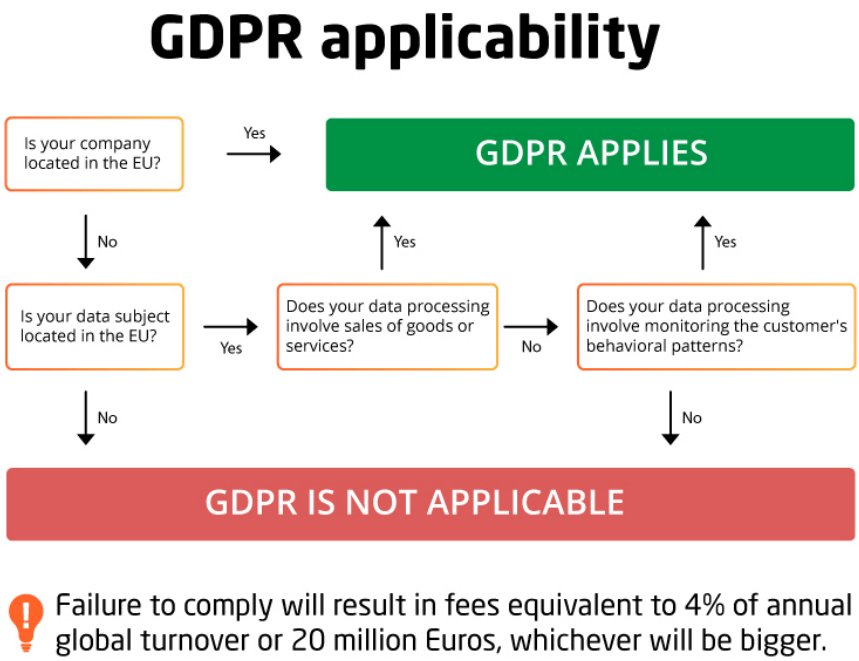
Horizontal scaling and GDPR-compliance

Once the ML algorithm is in place, the processed data becomes a valuable asset. This is even more true for personal data of end users, which must now comply with the GDPR requirements. In addition, while pursuing the main goal we often come across some unexpected applications of the ML solution, allowing to further enhance and improve the efficiency of the customer's business processes. We might implement these auxiliary systems per customer's request.

For example, we can deploy custom-tailored Big Data Analytics systems based on the implemented ML algorithms. We can train the customer's staff to use the systems and apply the results for brand promotion or comparison of the data usage efficiency with the competitors.



One of the biggest concerns, however, lies in the increasing importance of ensuring the security of storage for the processed data. We provide stress-testing of the customer's data security systems and help meet the levels required to comply with the GDPR. This is especially important as **failure to comply will result in fees equivalent to 4% of annual global turnover or 20 million Euros, whichever will be bigger.**



To summarize all the above, designing and building a Machine Learning solution can be a lengthy and effort-consuming process, highly-dependant on the skills and expertise of the developers. However, executing this task correctly results in an accurate and fast-performing solution for predictive analytics, a great support for your business decision-making.

Should you like to create a custom Machine Learning solution for your needs — IT Svit is ready to help!

About IT Svit

We help industry-leading companies in the US, EU and worldwide innovate and overcome their challenges. IT Svit does this using enterprise-grade technology solutions to drive value and ensure business continuity. We pride ourselves on delivering great user experience and brand advocacy.

IT Svit specializes in DevOps services, Big Data technology, Machine Learning, bespoke Blockchain platforms, full-cycle services for startups, web development and QA.