Train with Python, predict with C++

Pavel Filonov

About me

Pavel Filonov

Research Development Team Lead at Kaspersky Lab

Machine Learning everywhere!

- Mobile
- Embedded
- Automotive
- Desktops
- Games
- Finance
- Etc.

Image from [1]

Machine Learning face the real world

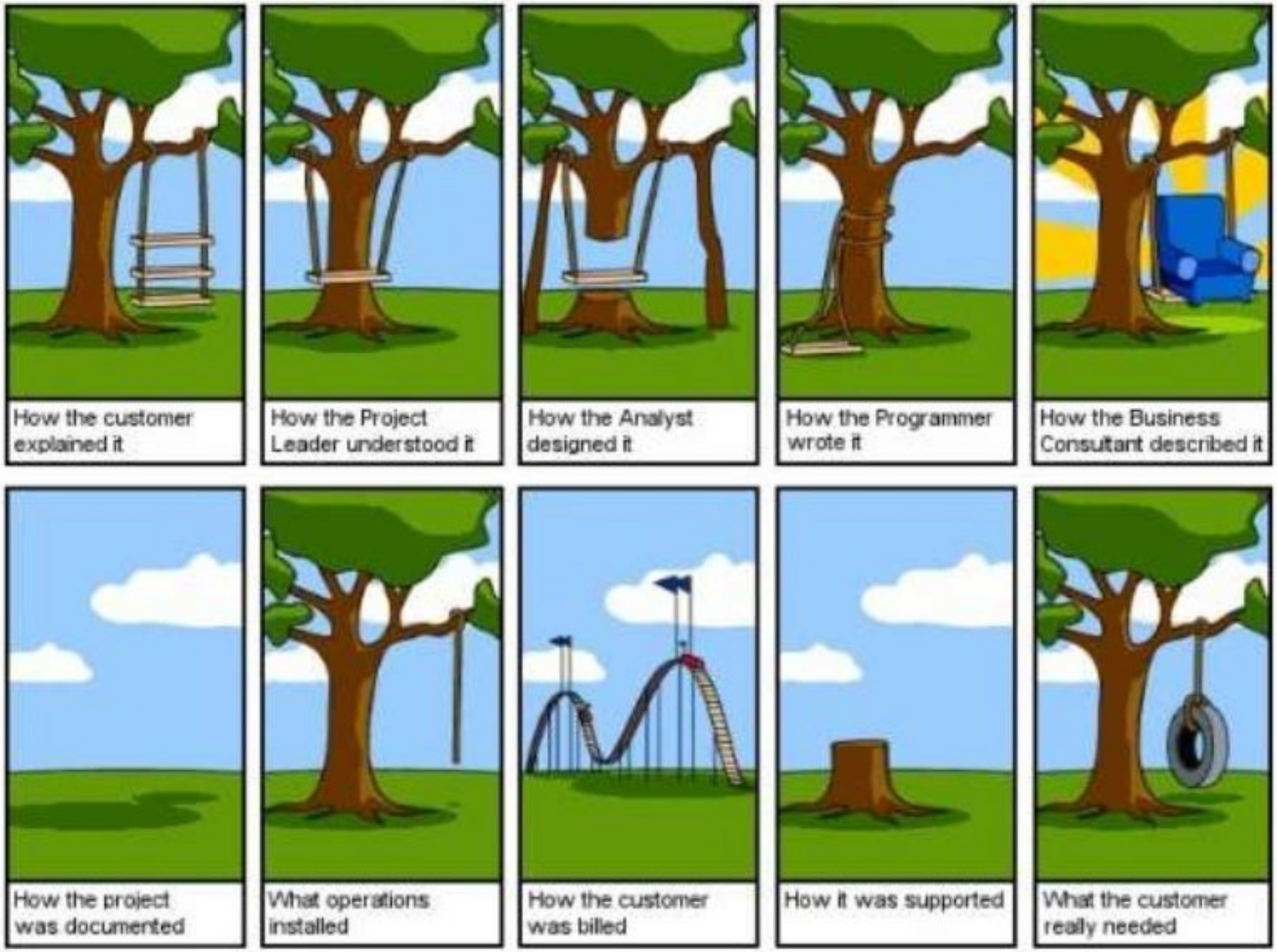


Image from [2]

CRISP-DM

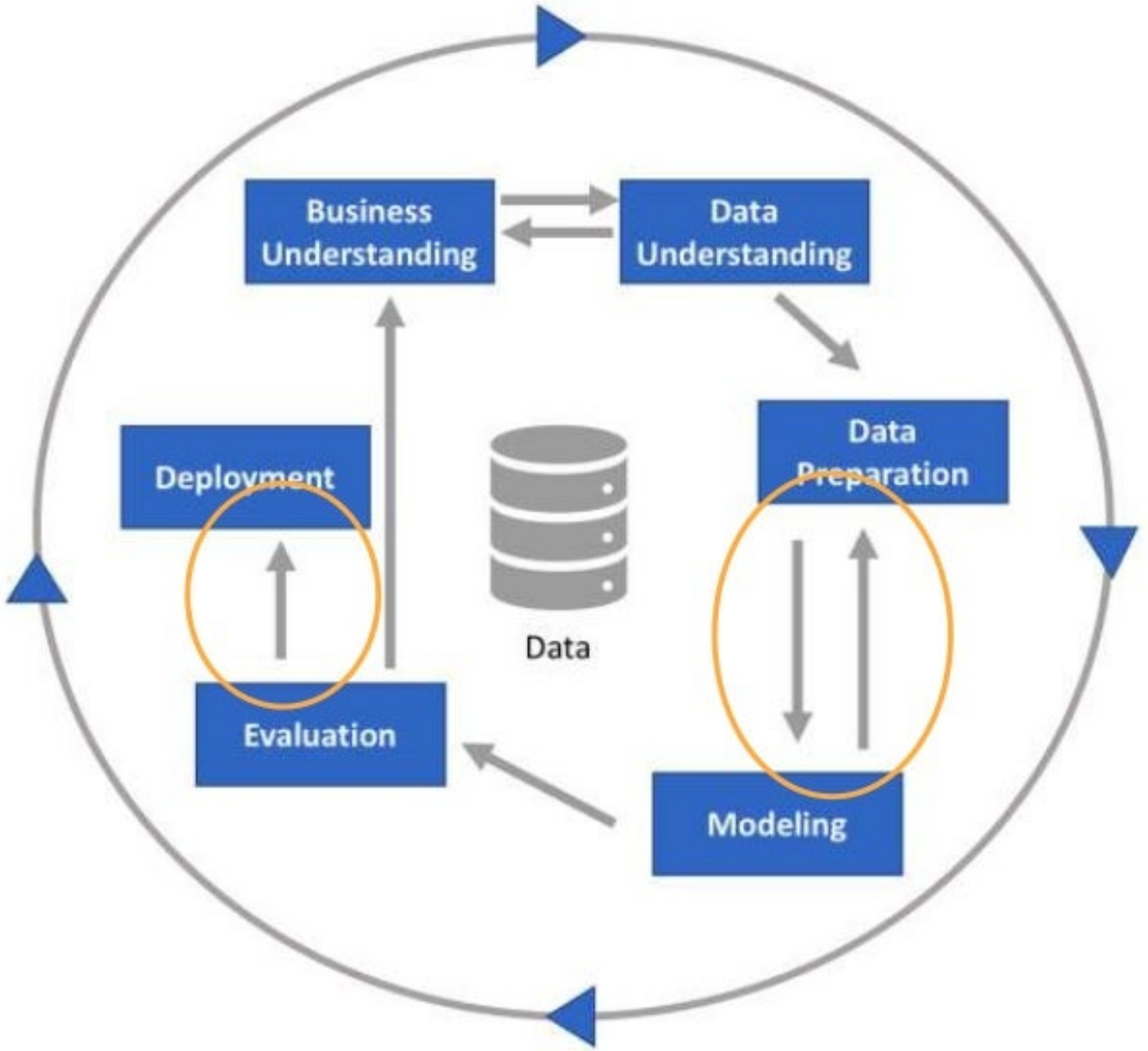


Image from [3]

Dream team

Developer

Data Scientist

Dream team – synergy way

Developer

Data Scientist

Research Developer

Dream team – process way

Developer

Data Scientist

Machine learning sample cases

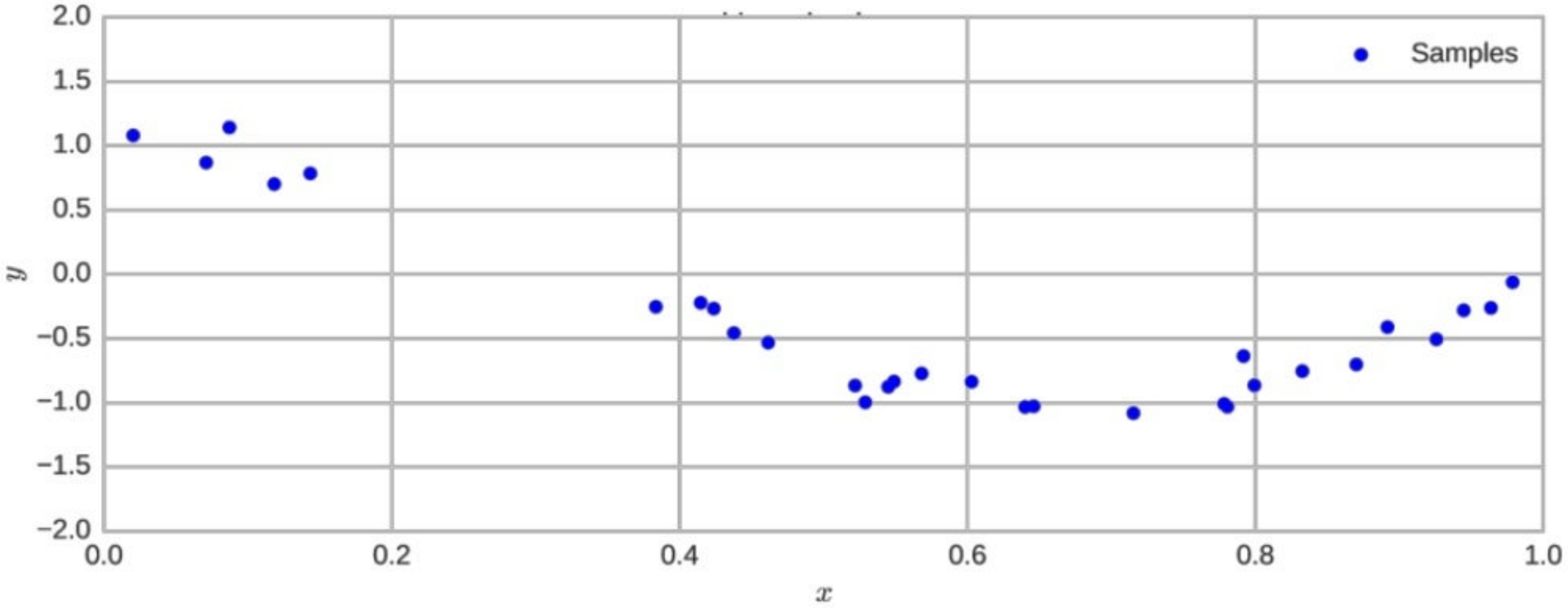
1. Energy efficiency prediction
2. Intrusion detection system
3. Image classification

Buildings Energy Efficiency

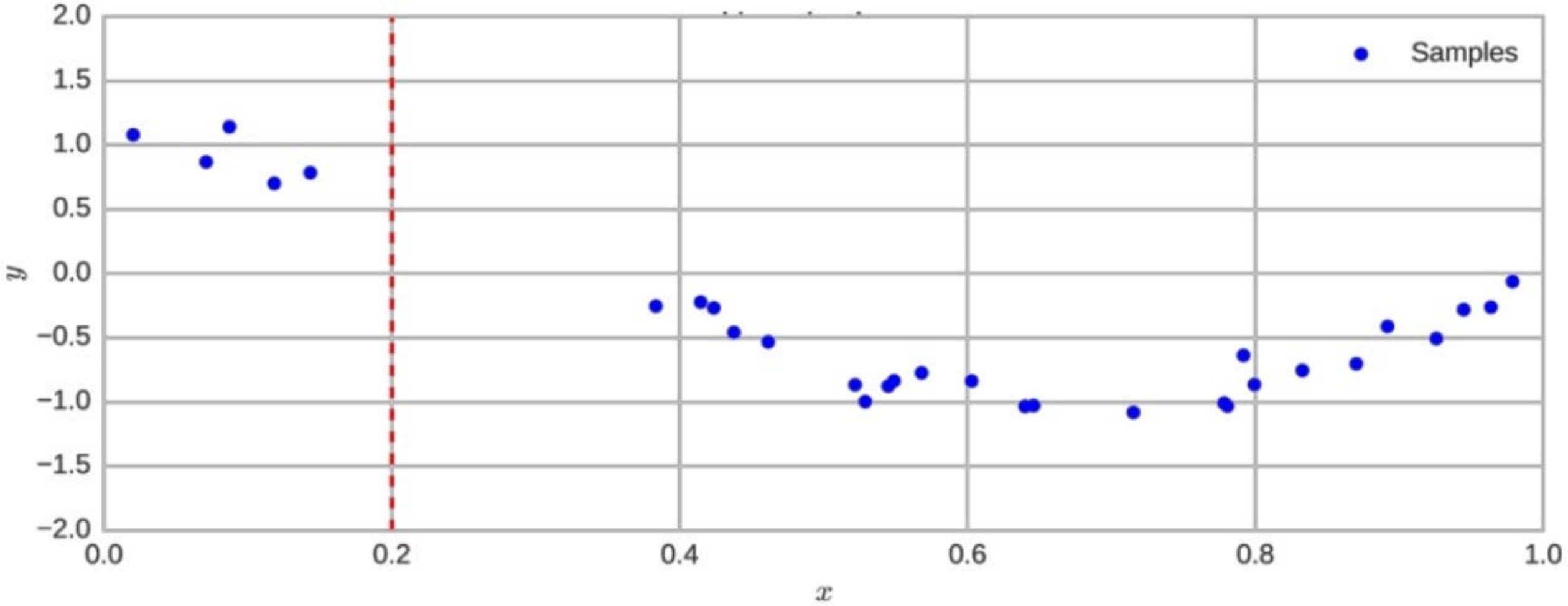
- Input attributes
  - Relative Compactness
  - Surface Area
  - Wall Area
  - etc.
- Outcomes
  - Heating Load

ref: [4]

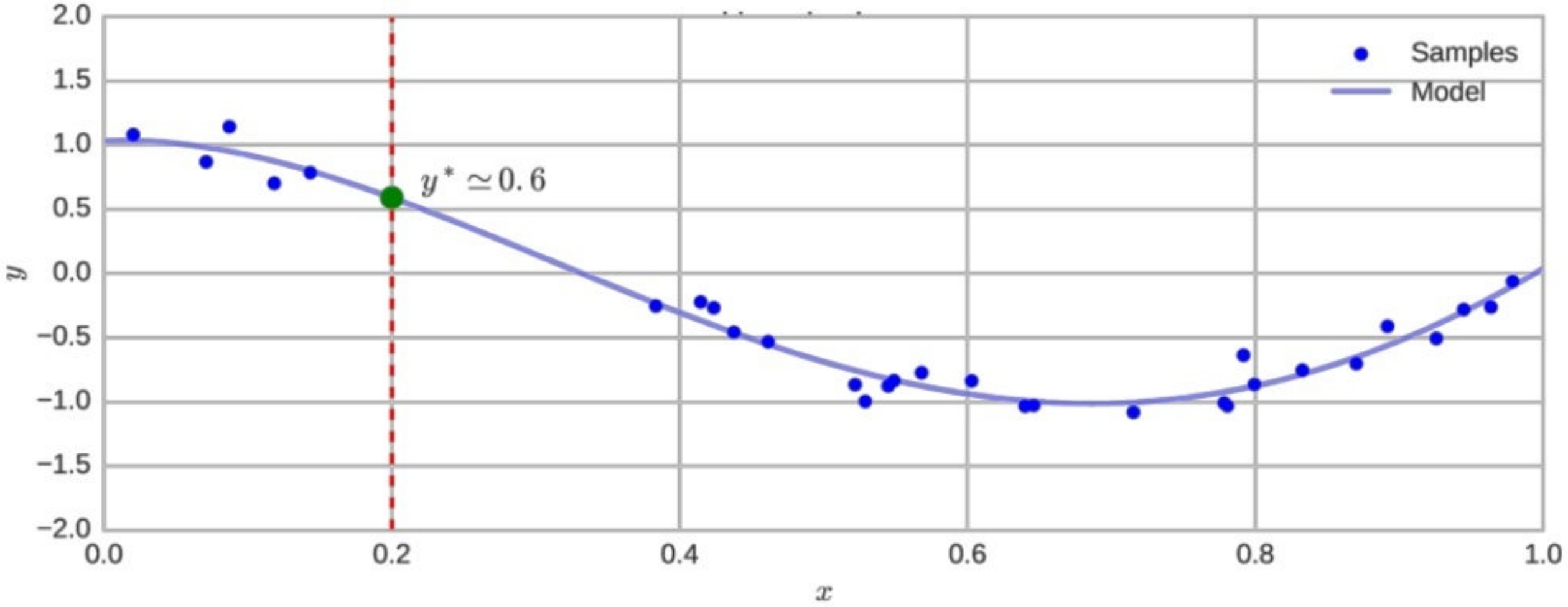
Regression problem



Regression problem



Regression problem



Quality metric

- Determination coefficient

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}},$$

$$SS_{tot} = \sum_i (y_i - \bar{y})^2, \qquad SS_{res} = \sum_i (y_i^* - \bar{y})^2, \qquad \bar{y} = \sum_i y_i$$

- $-1 < R^2 < 0$ – bad model
- $R^2 = 0$ – always predict mean ($\bar{y}$)
- $0 < R^2 < 1$ – useful model

Baseline model

- always predict mean
- $R^2 = 0$
- easy to develop

```
class Predictor {
public:
    using features = std::vector<double>;

    virtual ~Predictor() {};
    virtual double predict(const features&) const = 0;
};

class MeanPredictor: public Predictor {
public:
    MeanPredictor(double mean);


    double predict(const features&) const override { return mean_; }


protected:
    double mean_;
};
```

Linear regression

- predict $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_m x_m$,
  $\vec{x}$ - input, $\vec{\theta}$-model parameters
- $R^2 = 0.9122$
- use stdlib Luke

```
class LinregPredictor: public Predictor {
public:
    LinregPredictor(const std::vector<double>&);


    double predict(const features& feat) const override {
        assert(feat.size() + 1 == coef_.size());
        return  std::inner_product(feat.begin(), feat.end(), ++coef_.begin(),
                                   coef_.front());
    }

protected:
    std::vector<double> coef_;
};
```

Polynomial regression

- predict $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_m x_m +$
$+ \theta_{m+1} x_1^2 + \theta_{m+2} x_1 x_2 + \cdots + \theta_{\frac{m(m+1)}{2}} x_m^2$,

$\vec{x}$ - input, $\vec{\theta}$-model parameters

```
class PolyPredictor: public LinregPredictor {
public:
    using LinregPredictor::LinregPredictor;

    double predict(const features& feat) const override {
        features poly_feat{feat};
        const auto m = feat.size();
        poly_feat.reserve(m*(m+1)/2);
        for (size_t i = 0; i < m; ++i) {
            for (size_t j = i; j < m; ++j) {
                poly_feat.push_back(feat[i]*feat[j]);
            }
        }
        return LinregPredictor::predict(poly_feat);
    }
};
```

Integration testing

- you always have a lot of data for testing
- use python model output as expected values
- beware of floating point arithmetic problems

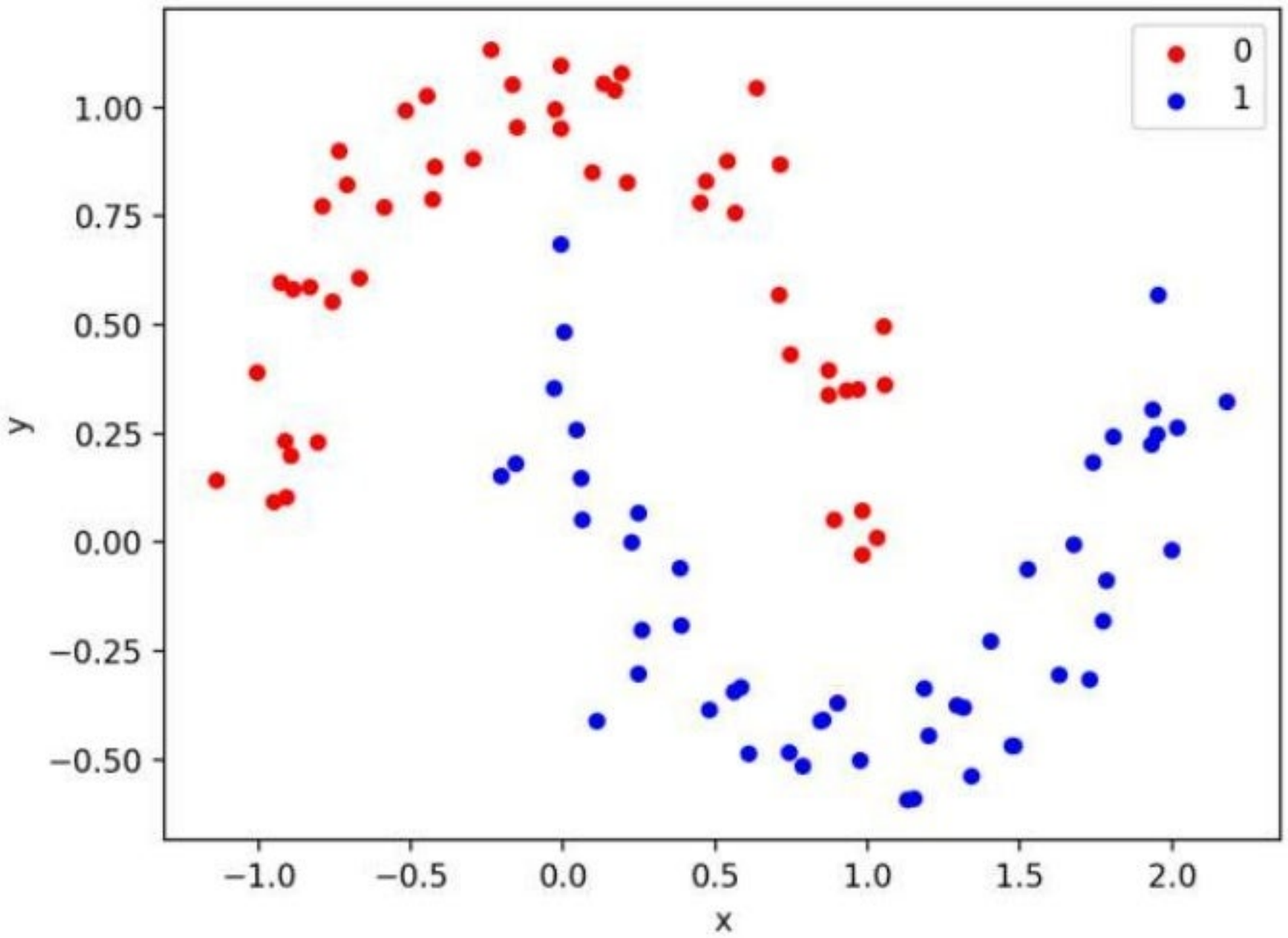
```
TEST(LinregPredictor, compare_to_python) {
    auto predictor = LinregPredictor{coef};
    double y_pred_expected = 0.0;

    std::ifstream test_data{"../train/test_data_linreg.csv"};
    while (read_features(test_data, features)) {
        test_data >> y_pred_expected;
        auto y_pred = predictor.predict(features);
        EXPECT_NEAR(y_pred_expected, y_pred, 1e-4);
    }
}
```

Intrusion detection system

- input - network traffic features
  - protocol_type
  - connection duration
  - src_bytes
  - dst_bytes
  - etc.
- Output
  - normal
  - network attack

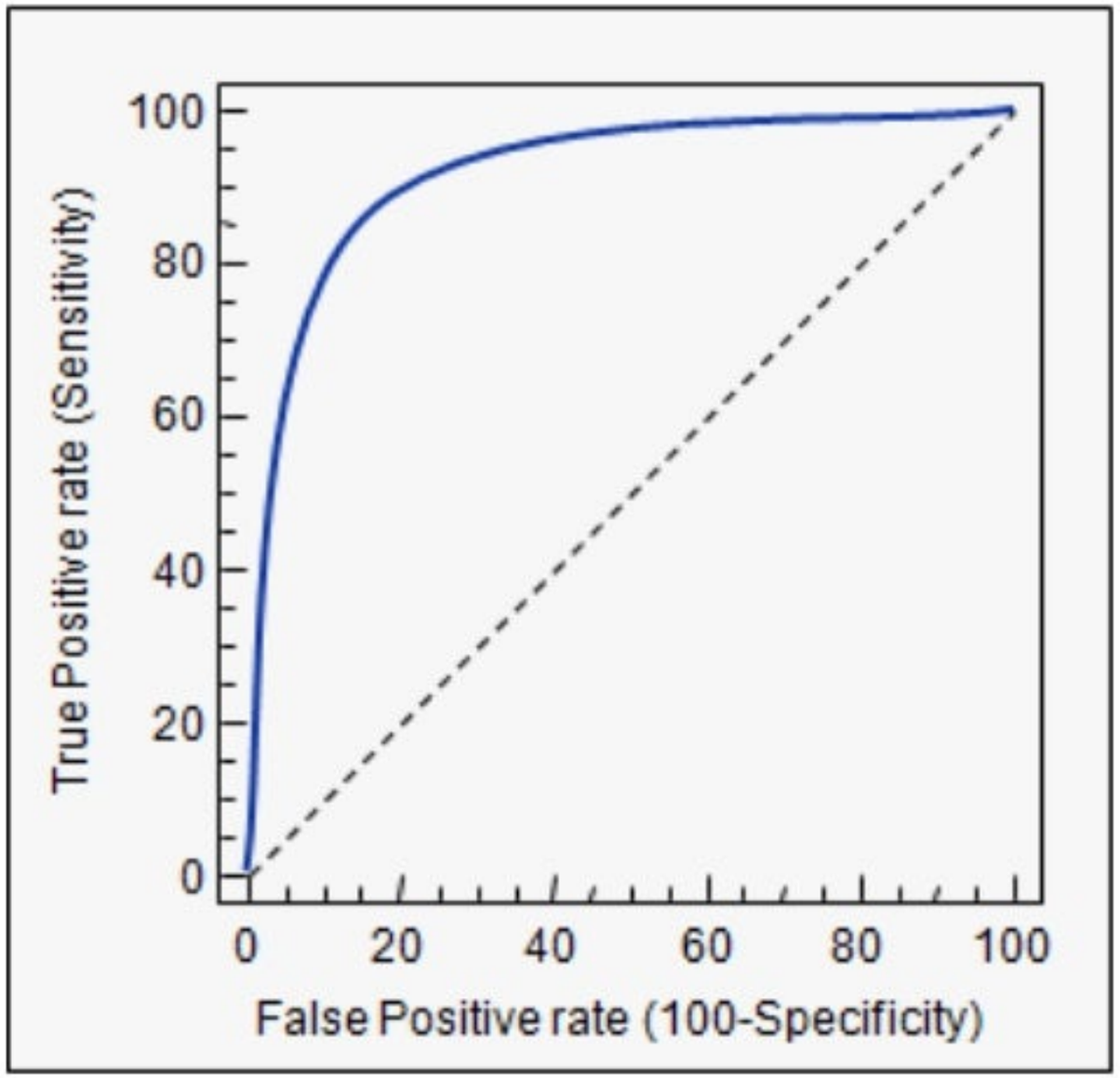
ref: [5]

Classification problem



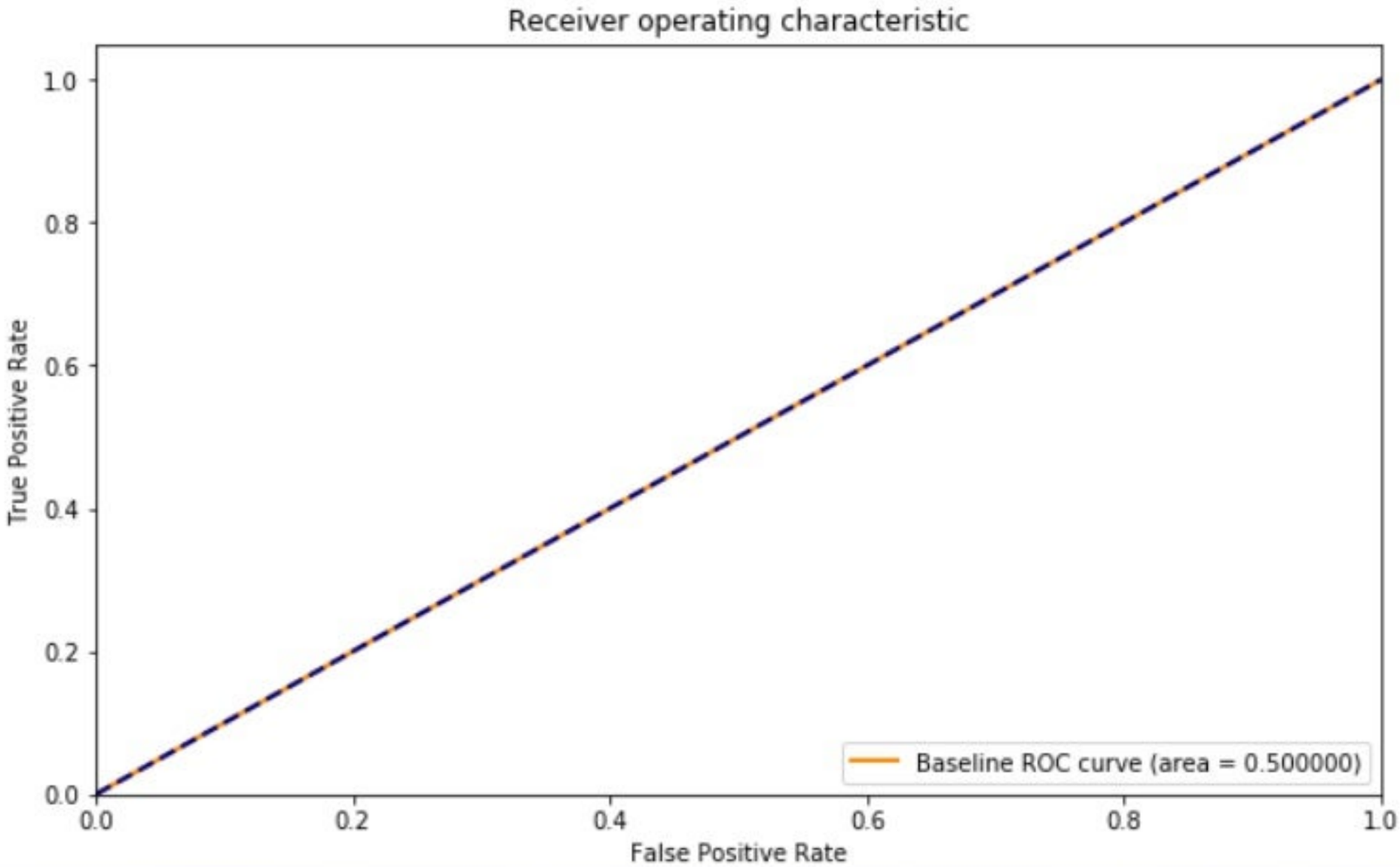
Quality metrics

- Receive operation characteristics (ROC) curve



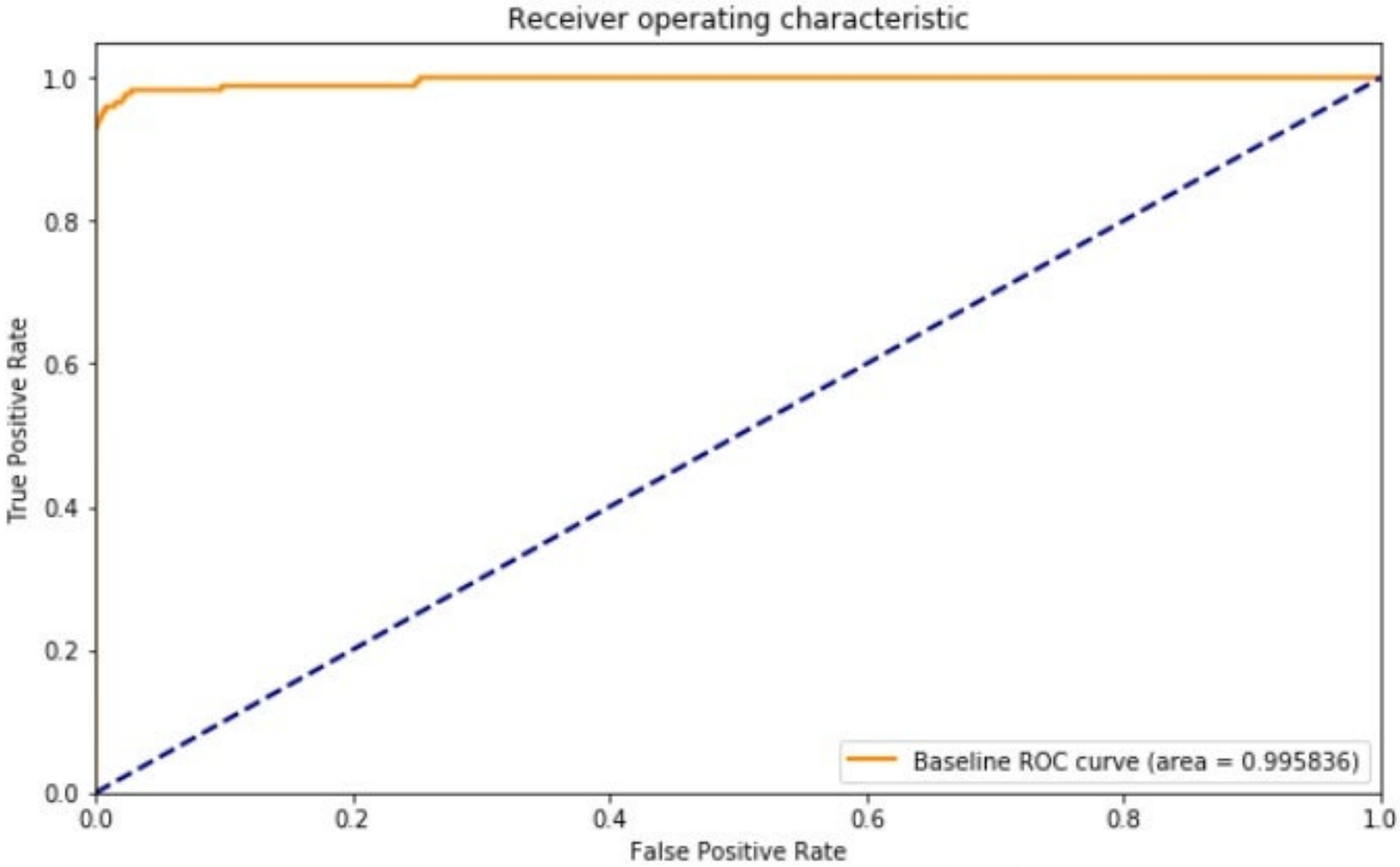
Baseline model

- always predict most frequent class
- ROC area under the curve = 0.5



Logistic regression

- $h_\theta(x) = \sigma(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$ - “probability” of positive class
- ROC area under the curve = 0.9958



Logistic regression

- easy to implement

```
template<typename T>
auto sigma(T z) {
    return 1/(1 + std::exp(-z));
}
```

```
class LogregClassifier: public BinaryClassifier {
public:
    float predict_proba(const features_t& feat) const override {
        auto z = std::inner_product(feat.begin(), feat.end(),
++coef_.begin(), coef_.front());
        return sigma(z);
    }


protected:
    std::vector<float> coef_;
};
```

Gradient boosting

- de facto standard universal method
- multiple well known C++ implementations with python bindings
  - XGBoost
  - LigthGBM
  - CatBoost
- each implementation has its own custom model format

CatBoost

- C API and C++ wrapper
- own build system (ymake)

```
class CatboostClassifier: public BinaryClassifier {
public:
    CatboostClassifier(const std::string& modepath);
    ~CatboostClassifier() override;

    double predict_proba(const features_t& feat) const override {
        double result = 0.0;
        if (!CalcModelPredictionSingle(model_, feat.data(), feat.size(),
                                       nullptr, 0, &result, 1)) {
            throw std::runtime_error{"CalcModelPredictionFlat error message:" +
                                     GetErrorString()};
        }
        return result;
    }
private:
    ModelCalcerHandle* model_;
}
```

CatBoost

- ROC-AUC = 0.9999

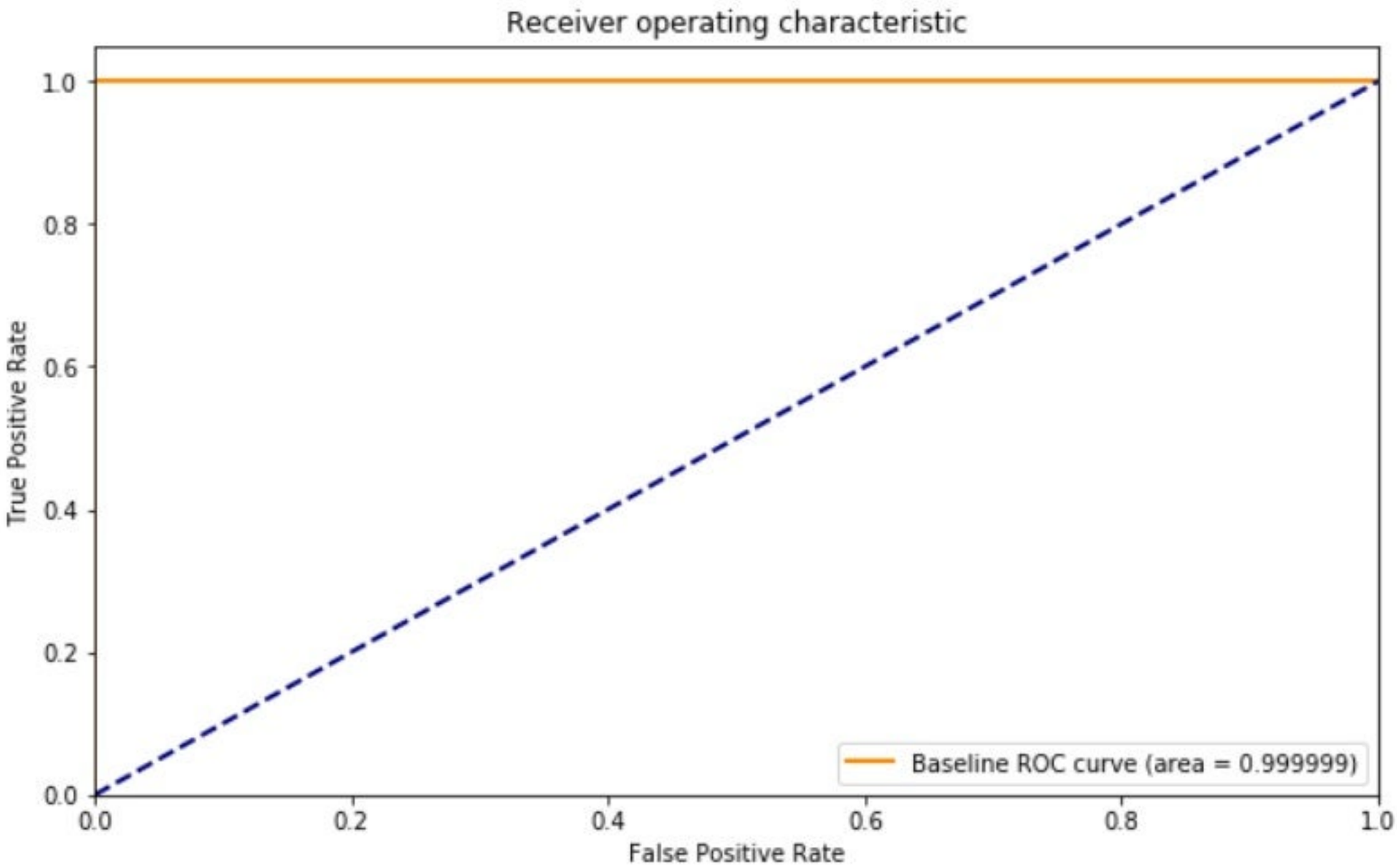
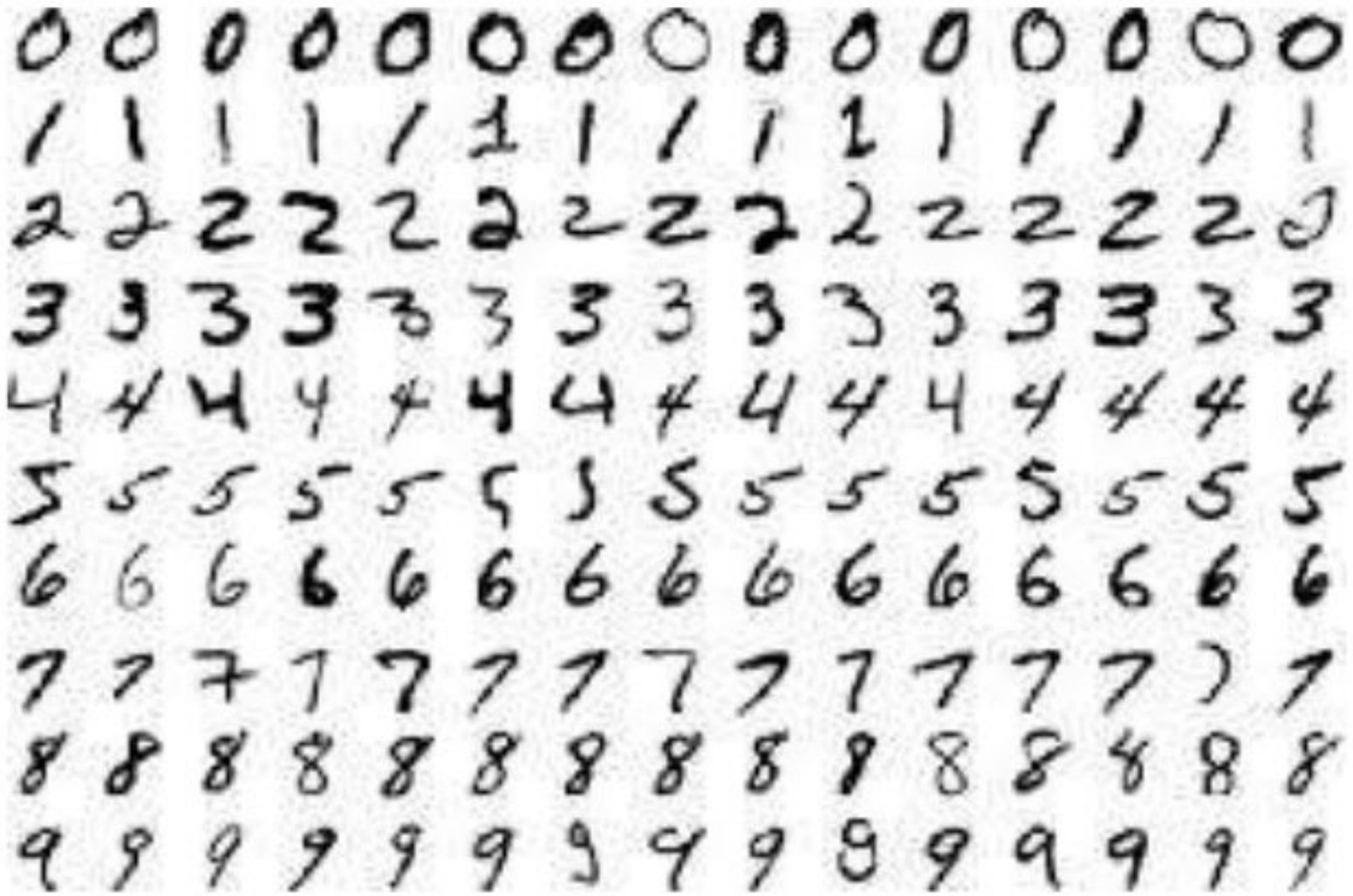


Image classification

- Handwritten digits recognizer – MNIST
- input – gray-scale pixels 28x28
- output – digit on picture (0, 1, … 9)



Multilayer perceptron

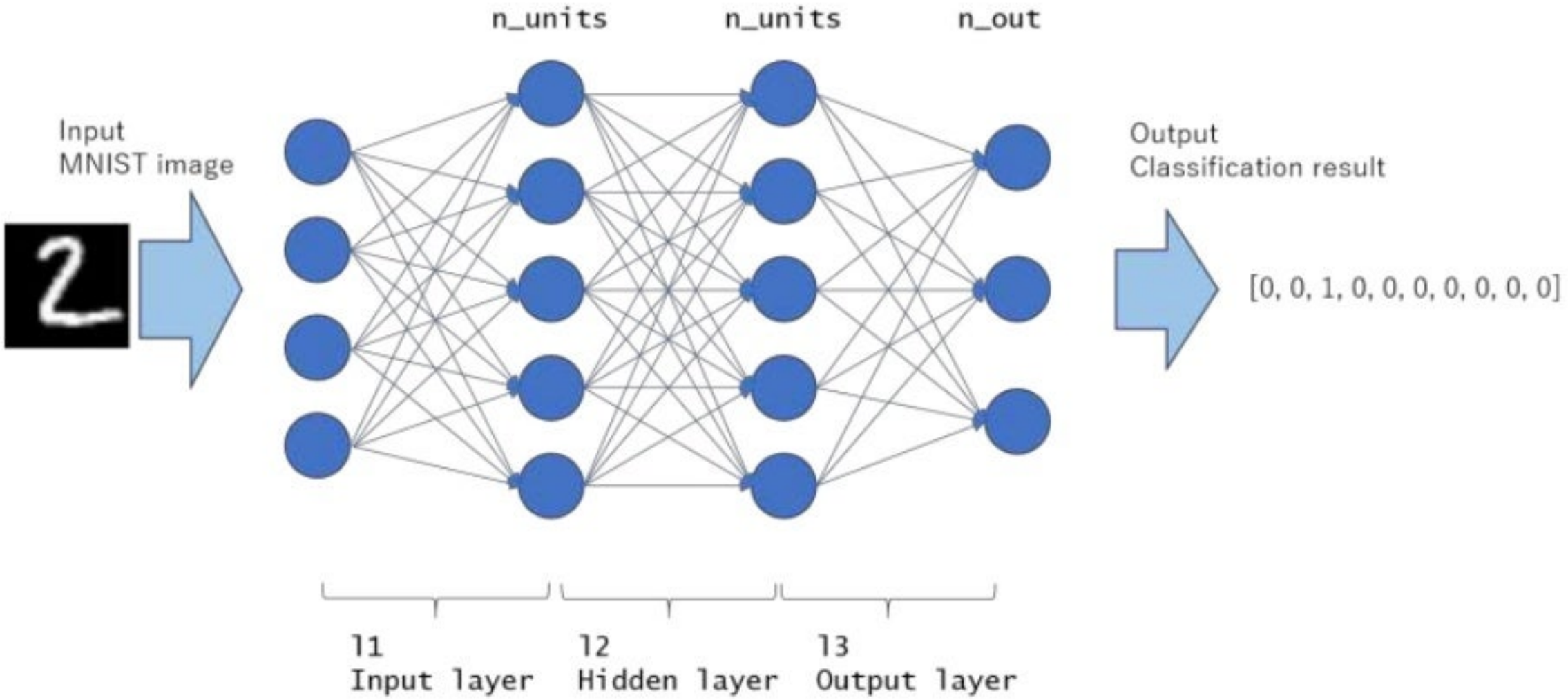
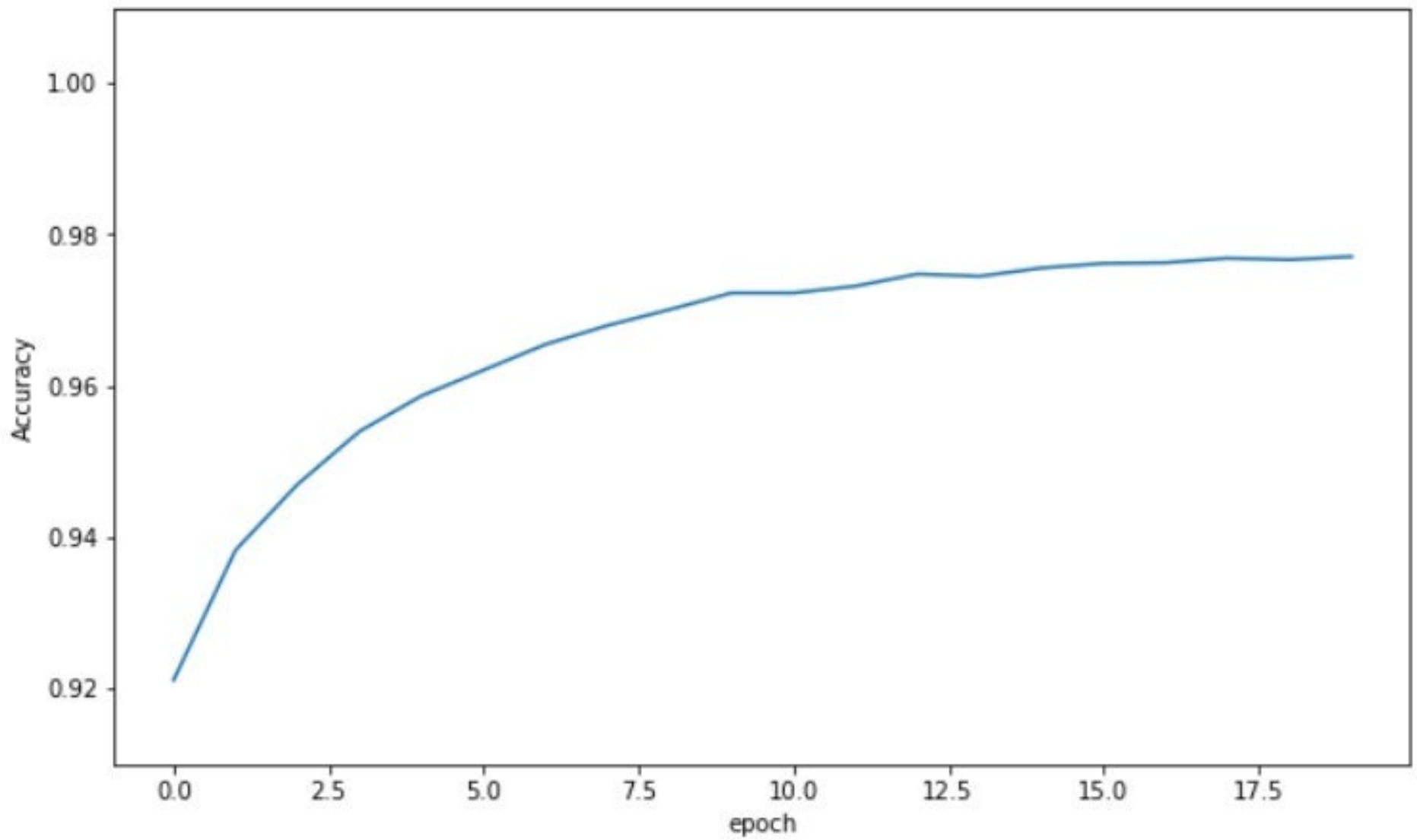


Image from: [6]

Quality metrics

- $Accuracy = \frac{correct}{total}$



Multilayer perceptron

- prediction – just a matrix multiplication

$$o_1 = \sigma(W_1 x)$$
$$o_2 = softmax(W_2 o_1)$$

```
auto MlpClassifier::predict_proba(const features_t& feat) const {
    VectorXf x{feat.size()};

    auto o1 = sigmav(w1_ * x);
    auto o2 = softmax(w2_ * o1);

    return o2;
}
```

Multilayer perceptron

- prediction – just a matrix multiplication

$$o_1 = \sigma(W_1 x)$$
$$o_2 = softmax(W_2 o_1)$$

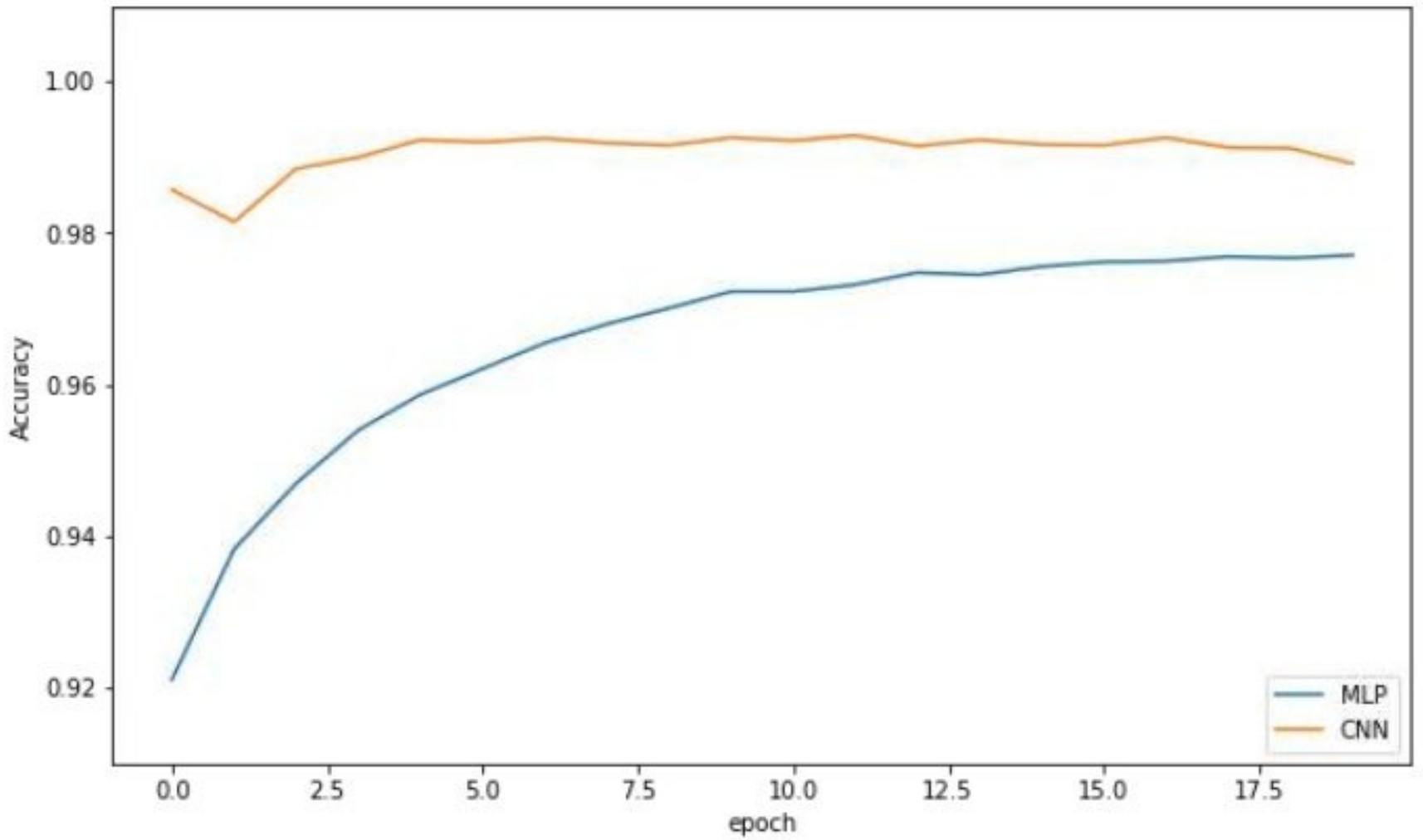
```
auto MlpClassifier::predict_proba(const features_t& feat) const {
    VectorXf x{feat.size()};

    auto o1 = sigmav(w1_ * x);
    auto o2 = softmax(w2_ * o1);

    return o2;
}
```

Convolutional networks

- State of the Art algorithms in image processing
- a lot of C++ implementation with python bindings
    - TensorFlow
    - Caffe
    - MXNet
    - CNTK

Tensorflow

- C++ API
- Bazel build system
- Hint – prebuild C API



Conclusion

- Don’t be fear of the ML
- Try simpler things first
- Get benefits from different languages

References

1. **Andrew Ng,** Machine Learning – coursera
2. **How to Get the Right Creative Requirements From Your Client**
3. **The Forgotten Step in CRISP-DM and ASUM-DM Methodologies**
4. **Energy efficiency Data Set**
5. **KDD Cup 1999**
6. **MNIST training with Multi Layer Perceptron**
7. **Code samples**

Many thanks!

Let’s talk

Pavel Filonov Pavel.Filonov@kaspersky.com +7(966)077-32-80