Dataset description

Chosen dataset: banknote authentication dataset.

Data were extracted from images that were taken from genuine and forged banknote-like specimens.

The target class shows whether the banknote is forged or not.

Number of entities:

1372

Features:

1. variance of Wavelet Transformed image (continuous)
2. skewness of Wavelet Transformed image (continuous)
3. curtosis of Wavelet Transformed image (continuous)
4. entropy of image (continuous)
5. is_forged (integer)

Changing feature space

The head of data before preprocessing:

```
data[:2]
```

	variance	skewness	curtosis	entropy	is_forged
0	3.6216	8.6661	-2.8073	-0.44699	0
1	4.5459	8.1674	-2.4586	-1.46210	0

All continuous features were divided on intervals by four distribution quantiles.

Dividing by quantiles

```
features = ['variance', 'skewness', 'curtosis', 'entropy']
for i in features:
    data[i] = pd.qcut(data[i], 4)
```

```
data = pd.get_dummies(data, columns = features)
```

```
features[0] = features[0][:3]

prefix = []
for i in features:
    prefix = prefix + [i + str(j) for j in range(4)]

for i, k in zip(data.columns[1:], (range(len(data.columns[1:])))):
                data.rename(columns={i: prefix[k]}, inplace=True)
```

The head of data after binarization:

```
data[:2]
```

	is_forged	var0	var1	var2	var3	skewness0	skewness1	skewness2	skewness3	curtosis0	curtosis1	curtosis2	curtosis3	entropy0	entropy1	e
0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	
1	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1	

Cross-Validation

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=10, shuffle=True, random_state=None)

for k, (train, test) in enumerate(kf.split(data)):
    data.iloc[train].to_csv('train' + str(k+1) + '.csv', index=False, header=False)
    data.iloc[test].to_csv('test'+str(k+1) + '.csv', index=False, header=False)
```

Used functions

Loading test and train. Composing train on plus and minus contexts

```
def load(i):
    train = pd.read_csv('train' + str(i) + '.csv' , sep=',', header=None)
    plus = train[train[0]==1]
    minus = train[train[0]==0]
    unknown = pd.read_csv('test' + str(i) + '.csv' , sep=',', header=None)

    return np.array(plus), np.array(minus), np.array(unknown)
```

Converting the context to the required format

```
attrib_names=list(data)
attrib_names
```

```
['is_forged',
 'var0',
 'var1',
 'var2',
 'var3',
 'skewness0',
 'skewness1',
 'skewness2',
 'skewness3',
 'curtosis0',
 'curtosis1',
 'curtosis2',
 'curtosis3',
 'entropy0',
 'entropy1',
 'entropy2',
 'entropy3']
```

```
def make_intent(example):
    return set([i + ':' + str(k) for i, k in zip(attrib_names, example) if k])
```

Algorithm1

The first algorithm is the easy one. We just calculate the normalized by context length intersection between example and context and assign the example the label of the class where the intersection was greater. The algorithm has time complexity $O(|train| \cdot |test|)$. And works fast, compared with others.

Algorithm1

```
def algorithm1(context_plus, context_minus, example):
    a = 0; b = 0
    eintent = make_intent(example)
    eintent.discard('is_forged:1')
    eintent.discard('is_forged:0')
    labels = {"positive": 0, "negative": 0}

    for e in context_plus:
        ei = make_intent(e)
        candidate_intent = ei & eintent
        a += len(candidate_intent)

    for e in context_minus:
        ei = make_intent(e)
        candidate_intent = ei & eintent
        b +=len(candidate_intent)

    a = a/len(context_plus)
    b = b/len(context_minus)

    if a > b:
        if example[0]:
            return "TP"
        return "FP"

    elif a < b:
        if example[0]:
            return "FN"
        return "TN"
    else:
        return "contradictory"
```

Algorithm 2

We have a big dataset. And this algorithm complexity is rather high($O(|train|^2 * |test|)$), so the calculations for whole trains would take several hours. But it still works great on randomly chosen 10% of each train.

Each object from the plus context 'votes' for a positive classification if its intersection with the example is not embedded in the descriptions from the minus context (and Vice versa).

An example is classified positively if the number of 'votes' for a positive classification prevails (and Vice versa).

Algorithm 2

```
def algorithm2(context_plus, context_minus, example):

    l = list(range(len(context_plus)))
    random.shuffle(l)
    context_plus = context_plus[l[:55]]
    l = list(range(len(context_minus)))
    random.shuffle(l)
    context_minus = context_minus[l[:68]]

    eintent = make_intent(example)
    eintent.discard('is_forged:1')
    eintent.discard('is_forged:0')
    labels = {"positive": 0, "negative": 0}
    for e in context_plus:
        ei = make_intent(e)
        candidate_intent = ei & eintent
        closure = [make_intent(i).issuperset(candidate_intent) for i in context_minus]

        if sum(closure)==0:
            labels["positive"] += 1
    for e in context_minus:
        ei = make_intent(e)
        candidate_intent = ei & eintent
        closure = [make_intent(i).issuperset(candidate_intent) for i in context_plus]

        if sum(closure)==0:
            labels["negative"] += 1

    labels["positive"] = labels["positive"]/len(context_plus)
    labels["negative"] = labels["negative"]/len(context_minus)

    if labels["positive"] > labels["negative"]:
        if example[0]:

            return "TP"
        return "FP"
    elif labels["positive"] < labels["negative"]:
        if example[0]:
            return "FN"
        return "TN"
    elif labels["positive"] == labels["negative"]:
        return "contradictory"
```

Quality metrics

Metrics

```
def accuracy(r):
    return float(r["TP"] + r["TN"]) / max(1, r["TP"] + r["TN"] + r["FP"] + r["FN"] + r["contradictory"])

def precision(r):
    return float(r["TP"]) / max(1, r["TP"] + r["FP"])

def recall(r):
    return float(r["TP"]) / max(1, r["TP"] + r["FN"])

def results(r):
    metrics = {}
    metrics["accuracy"] = accuracy(r)
    metrics["precision"] = precision(r)
    metrics["recall"] = recall(r)
    return metrics
```

Written function for algorithm launch:

Algorithm Launch

```
def summary(algorithm_name):
    # time on
    import timeit
    start = timeit.default_timer()

    acc = 0
    prec = 0
    rec = 0
    for index in range(1,11):
        (iplus, iminus, iunknown) = load(index)
        cv_res = {
            "TP": 0,
            "TN": 0,
            "FP": 0,
            "FN": 0,
            "contradictory": 0,
            }
        for elem in iunknown:
            pin = algorithm_name(iplus, iminus, elem)
            cv_res[pin] += 1

        res = results(cv_res)
        acc += res['accuracy']
        prec += res['precision']
        rec += res['recall']

    # find mean results for cross-validation
    acc = acc/10
    prec = prec/10
    rec = rec/10

    # time off
    stop = timeit.default_timer()
    time = stop - start

    return acc, prec, rec, time
```

Results

```
(a1,p1,r1,time1) = summary(algorithm1)
print('Accuracy: '+str(a1*100)+'%')
print('Precision: '+str(p1*100)+'%')
print('Recall: '+str(r1*100)+'%')
print('Time of algorithm work: '+str(time1))
```

```
Accuracy: 87.3135512535703%
Precision: 84.07073506887184%
Recall: 88.63998912627322%
Time of algorithm work: 27.214702186000068
```

```
(a2,p2,r2,time2) = summary(algorithm2)
print('Accuracy: '+str(a2*100)+'%')
print('Precision: '+str(p2*100)+'%')
print('Recall: '+str(r2*100)+'%')
print('Time of algorithm work: '+str(time2))
```

```
Accuracy: 94.2425684967735%
Precision: 94.04607271891679%
Recall: 96.66493532334384%
Time of algorithm work: 173.96348000800208
```

We see that accuracy and precision both are better for the second algorithm, but we should look at the time and remember that the second algorithm was realized only for 10% of each train but even with this privilege it is much worse in time.