

Machine Learning Zoomcamp

Session #1.2

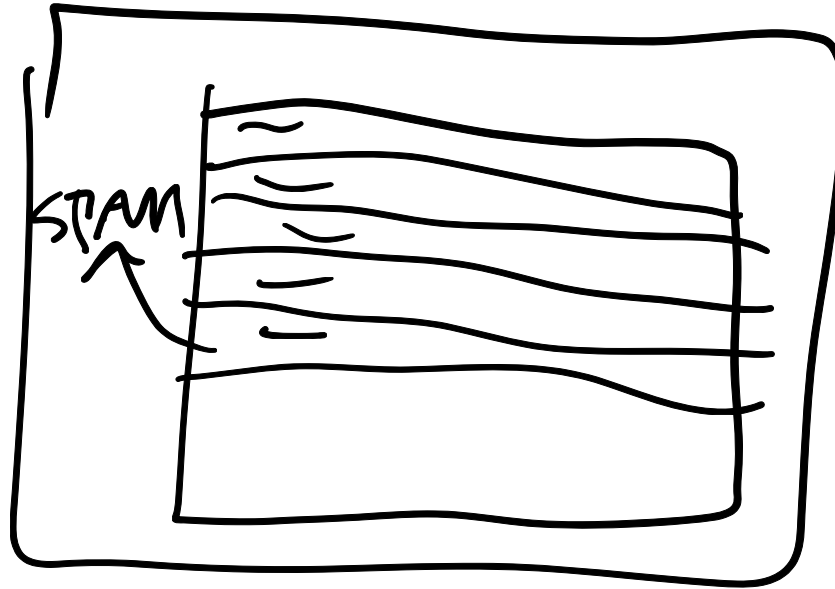
Machine Learning vs Rule-Based Systems

DataTalks.Club

Session #1.2: Plan

- A rule-based system for spam detection
- Using ML for spam detection
- Extracting features for ML

Email system



Spam

Subject: Get 50% off now
From: promotions@online.com

When
spam
you
click
Pay
with

Subject: URGENT: tax review
From: tax@online.com

Your tax review is pending acceptance. Review within 24 hours:

<https://taxes.we-are-legit.com>

Tax office.

Rules

- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- Otherwise, “good email”

Code

```
def detect_spam(email):  
    if email.sender == 'promotions@online.com':  
        return SPAM  
    if contains(email.title, ['tax', 'review']) and  
        domain(email.sender, 'online.com'):  
        return SPAM  
    return GOOD
```

More

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please **deposit** \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

Rules

- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- **If body contains a word “deposit” then “spam”**
- Otherwise, “good email”

Code

```
def detect_spam(email):  
    if email.sender == 'promotions@online.com':  
        return SPAM  
    if contains(email.title, ['tax', 'review']) and  
        domain(email.sender, 'online.com'):  
        return SPAM  
    if contains(email.body, ['deposit']):  
        return SPAM  
    return GOOD
```

More

Subject: Totally legit email

From: pedro@gmail.com

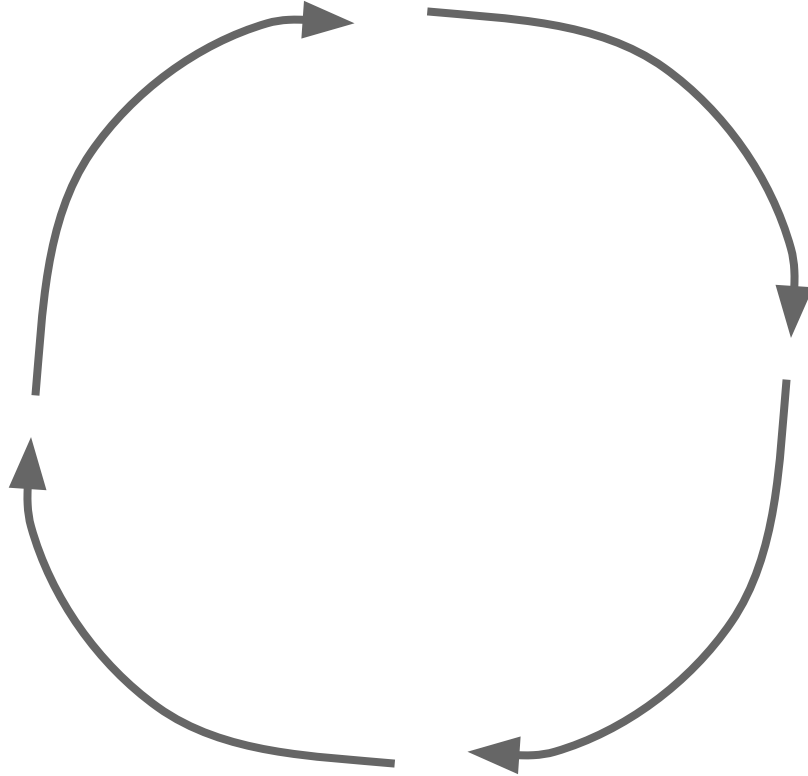
I transferred \$50 to you one year ago, and now I'm moving out.
Please refund my deposit.

Pedro.

Rules

- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- If body contains a word “deposit”
 - If sender domain is “test.com” then “spam”
 - If body \geq 100 words then spam
- Otherwise, “good email”

Repeat



```

74     return self._type_spec_class(component_specs, self.metadata)
75
76     def __repr__(self):
77         return '%s(%r, %r)' % (type(self).__name__, self.components, self.metadata)
78
79     def __eq__(self, other):
80         return (type(self) is type(other) and
81               self.components == other.components and
82               self.metadata == other.metadata)
83
84
85 # Another test CompositeTensor class. 'tf.nest' should treat different CT
86 # classes as different structure types (e.g. for assert_same_structure).
87 class CTSpec2(CTSpec):
88     pass
89
90
91 class CT2(CT):
92     _type_spec_class = CTSpec2
93
94
95 @test_util.run_all_in_graph_and_eager_modes
96 class CompositeTensorTest(test_util.TensorFlowTestCase, parameterized.TestCase):
97
98     @parameterized.parameters([
99         ('structure': CT(0),
100          'expected': [0],
101          'paths': [['CT']],),
102         ('structure': CT('a'),
103          'expected': ['a'],
104          'paths': [['CT']],),
105         ('structure': CT(['a', 'b', 'c']),
106          'expected': ['a', 'b', 'c'],
107          'paths': [['CT', 0], ('CT', 1), ('CT', 2)]),
108         ('structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
109          'expected': ['a', 'b', 'c'],
110          'paths': [['CT', 'x'], ('CT', 'y'), ('CT', 'z')]),
111         ('structure': [{'k1': CT('a')}, CT({'b': 'x': CT('y': 'c')})]),
112         ('expected': ['a', 'b', 'c'],
113          'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT', 1, 'x', 'CT', 'y')]),
114         ('structure': CT(0),
115          'expand_composites': False,
116          'expected': [CT(0)],
117          'paths': [0]),
118         ('structure': [{'k1': CT('a')}, CT({'b': 'x': CT('y': 'c')})]),
119         'expand_composites': False,
120         'expected': [CT('a'), CT('b', {'x': CT('y': 'c')})],
121         'paths': [(0, 'k1'), (1,)]),
122     ]) # pyformat: disable
123     def testNestFlatten(self, structure, expected, paths, expand_composites=True):
124         result = nest.flatten(structure, expand_composites=expand_composites)
125         self.assertEqual(result, expected)
126
127         result_with_paths = nest.flatten_with_tuple_paths(
128             structure, expand_composites=expand_composites)
129         self.assertEqual(result_with_paths, list(zip(paths, expected)))
130
131         string_paths = ['/'.join(str(p) for p in path) for path in paths] # pylint: disable=g-complex-comprehension
132         result_with_string_paths = nest.flatten_with_joined_string_paths(
133             structure, expand_composites=expand_composites)
134         self.assertEqual(result_with_string_paths,
135                          list(zip(string_paths, expected)))
136
137         flat_paths_result = list(
138             nest.yield_flat_paths(structure, expand_composites=expand_composites))
139         self.assertEqual(flat_paths_result, paths)
140
141     @parameterized.parameters([
142         ('s1': [1, 2, 3],
143          's2': [CT(['a', 'b']), 'c', 'd'],
144          'expand_composites': False,
145          'expected': [CT(['a', 'b']), 'c', 'd'],
146          'paths': [(0,), (1,), (2,)]),
147     ])

```

```

74     return self._type_spec_class(component_specs, self.meta
75
76     def __repr__(self):
77         return '%s(%r, %r)' % (type(self).__name__, self.components)
78
79     def __eq__(self, other):
80         return (type(self) is type(other) and
81                 self.components == other.components and
82                 self.metadata == other.metadata)
83
84 # Another test CompositeTensor class. 'tf.nn' should be
85 # classes as different structure types (e.g. for assert_is_
86 class CTSpec2(CTSpec):
87     pass
88
89
90 class CT2(CT):
91     _type_spec_class = CTSpec2
92
93
94 @test_util.run_all_in_graph_and_eager_modes
95 class CompositeTensorTest(test_util.TensorFlowTestCase,
96                             parameterized.TestCase):
97
98     @parameterized.parameters([
99         ('structure': CT(0),
100          'expected': [0],
101          'paths': [['CT']],
102          'structure': CT('a'),
103          'expected': ['a'],
104          'paths': [['CT']],
105          ('structure': CT(['a', 'b', 'c']),
106           'expected': ['a', 'b', 'c'],
107           'paths': [['CT', 0], ['CT', 1], ['CT', 2]],
108           'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
109           'expected': ['a', 'b', 'c'],
110           'paths': [['CT', 'x'], ['CT', 'y'], ['CT', 'z']],
111           ('structure': [['k1': CT('a')], CT(['b', 'x': CT(
112               'expected': ['a', 'b', 'c'],
113               'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT',
114                  ('structure': CT(0),
115                   'expand_composites': False,
116                   'expected': [CT(0)],
117                   'paths': [{}]),
118                   ('structure': [['k1': CT('a')], CT(['b', 'x': CT(
119                       'expand_composites': False,
120                       'expected': [CT('a'), CT(['b', 'x': CT(
121                           'paths': [(0, 'k1'), (1,)],
122                       ])) # pyformat: disable
123
124     def testNestFlatten(self, structure, expected, paths,
125                         result = nest.flatten(structure, expand_composites=ex
126                         self.assertEqual(result, expected)
127
128     result_with_paths = nest.flatten_with_tuple_paths(
129         structure, expand_composites=expand_composites)
130     self.assertEqual(result_with_paths, list(zip(paths,
131
132     string_paths = ['/'.join(str(p) for p in path) for
133     result_with_string_paths = nest.flatten_with_joined_s
134     structure, expand_composites=expand_composites)
135     self.assertEqual(result_with_string_paths,
136                       list(zip(string_paths, expected)))
137
138     flat_paths_result = list(
139         nest.yield_flat_paths(structure, expand_composit
140     self.assertEqual(flat_paths_result, paths)
141
142     @parameterized.parameters([
143         ('s1': [1, 2, 3],
144          's2': [CT(['a', 'b']), 'c', 'd'],
145          'expand_composites': False,
146          'expected': [CT(['a', 'b']), 'c', 'd'],
147          'paths': [(0,), (1,), (2,)]],
148
149
150     expand_composites = raise,
151     'expected': [CT(['a', 'b']), 'c', 'd'],
152     'paths': [(0,), (1,), (2,)]],
153     's1': [CT([1, 2, 3])],
154     's2': [5],
155     'expand_composites': False,
156     'expected': [5],
157     'paths': [(0,)]],
158     ('s1': [[CT([9, 9, 9])], 999, ('y': CT([9, 9])],
159      's2': [[CT([1, 2, 3])], 100, ('y': CT([CT([4, 5]), 6])],
160      'expand_composites': False,
161      'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
162      'paths': [(0, 0), (1,), (2, 'y')],
163      ('s1': [[CT([9, 9, 9])], 999, ('y': CT([CT([9, 9]), 9])],
164       's2': [[CT([1, 2, 3])], 100, ('y': CT([5, 6])],
165       'expand_composites': False,
166       'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
167       'paths': [(0, 0), (1,), (2, 'y')],
168   ]) # pyformat: disable
169
170     def testNestFlattenTo(self, s1, s2, expected, paths,
171                           expand_composites=True):
172         result = nest.flatten_to(s1, s2, expand_composites=expand_composites)
173         self.assertEqual(expected, result)
174
175     result_with_paths = nest.flatten_with_tuple_paths_to(
176         s1, s2, expand_composites=expand_composites)
177     self.assertEqual(result_with_paths, list(zip(paths, expected)))
178
179     @parameterized.parameters([
180         ('structure': CT(0),
181          'sequence': [5],
182          'expected': CT(5)),
183         ('structure': CT(['a', 'b', 'c']),
184          'sequence': ['a', CT(['b']), 'x': 'y']),
185          'expected': CT(['a', CT(['b']), 'x': 'y']),
186          ('structure': [['k1': CT(['a']), CT(['b', 'x': CT(
187              'sequence': ['a', 'b', 'c'],
188              'expected': [['k1': CT('a'), CT(['b', 'x': CT(
189              ('structure': [['k1': CT('a'), CT(['b', 'x': CT(
190                  'sequence': ['a', 'b'],
191                  'expand_composites': False,
192                  'expected': [['k1': 'a', 'b']],
193                  ('structure': CT(0, metadata='abc'),
194                   'sequence': [5],
195                   'expected': CT(5, metadata='abc')],
196               ])) # pyformat: disable
197
198     def testNestPackSequenceAs(self,
199                                structure,
200                                sequence,
201                                expected,
202                                expand_composites=True):
203         result = nest.pack_sequence_as(
204             structure, sequence, expand_composites=expand_composites)
205         self.assertEqual(result, expected)
206
207     @parameterized.parameters([
208         ('s1': CT('abc'), 's2': CT('xyz')),
209         ('s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])),
210         ('s1': [1, CT([10]), CT(200, metadata='xyz')],
211          's2': [0, CT([55]), CT(100, metadata='xyz')]),
212     ]) # pyformat: disable
213
214     def testNestAssertSameStructure(self, s1, s2, expand_composites=True):
215         nest.assert_same_structure(s1, s2, expand_composites=expand_composites)
216         nest.assert_shallow_structure(s1, s2, expand_composites=expand_composites)
217
218     @parameterized.parameters([
219         ('s1': CT(0), 's2': CT(['x'])),
220         ('s1': CT([1]), 's2': CT([1, 2])),
221         ('s1': CT(['a', 'b', 'c']), 's2': CT(['y': 1])),
222         ('s1': CT(0), 's2': CT(0, metadata='xyz')),
223         ('s1': CT(0, metadata='xyz'), 's2': CT(0)),
224         ('s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')),
225         ('s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])),
226         ('s1': [1, CT(['a']), CT('b', metadata='xyz')],
227

```

```

74     return self._type_spec_class(component_specs, self.meta
75
76     def __repr__(self):
77         return '%s(%s, %s)' % (type(self).__name__, self.comp
78
79     def __eq__(self, other):
80         return (type(self) is type(other) and
81                 self.components == other.components and
82                 self.metadata == other.metadata)
83
84
85 # Another test CompositeTensor class. 'tf.nnst' should t
86 # classes as different structure types (e.g. for assert_s
87 class CTSpec(CTSpec):
88     pass
89
90
91 class CT2(CT):
92     _type_spec_class = CTSpec2
93
94
95 @test_util.run_all_in_graph_and_eager_modes
96 class CompositeTensorTest(test_util.tensor_flow_test_case, p
97
98     @parameterized.parameters([
99         ('structure': CT(0),
100          'expected': [0],
101          'paths': [['CT']],
102          'structure': CT('a'),
103          'expected': ['a'],
104          'paths': [['CT']],
105          ('structure': CT(['a', 'b', 'c']),
106           'expected': ['a', 'b', 'c'],
107           'paths': [['CT', 0], ['CT', 1], ['CT', 2]]],
108          ('structure': CT(['x': 'a', 'y': 'b', 'z': 'c']),
109           'expected': ['a', 'b', 'c'],
110           'paths': [['CT', 'x'], ['CT', 'y'], ['CT', 'z']]),
111          ('structure': [['k1': CT('a')], CT(['b', 'x': CT(
112              'expected': ['a', 'b', 'c'],
113              'paths': [[0, 'k1', 'CT'], [1, 'CT', 0], [1, 'CT',
114              ('structure': CT(0),
115               'expand_composites': False,
116               'expected': [CT(0)],
117               'paths': [{}]],
118               ('structure': [['k1': CT('a')], CT(['b', 'x': CT(
119                   'expand_composites': False,
120                   'expected': [CT(0)],
121                   'paths': [[0, 'k1'], [1, 1]]],
122               ]) # pyformat: disable
123     def testNestFlatten(self, structure, expected, paths):
124         result = nest.flatten(structure, expand_composites=ex
125         self.assertEqual(result, expected)
126
127         result_with_paths = nest.flatten_with_tuple_paths(
128             structure, expand_composites=expand_composites)
129         self.assertEqual(result_with_paths, list(zip(paths, e
130
131         string_paths = ['/'.join(str(p) for p in path) for pa
132         result_with_string_paths = nest.flatten_with_joined_s
133         structure, expand_composites=expand_composites)
134         self.assertEqual(result_with_string_paths,
135             list(zip(string_paths, expected)))
136
137         flat_paths_result = list(
138             nest.yield_flat_paths(structure, expand_composit
139         self.assertEqual(flat_paths_result, paths)
140
141     @parameterized.parameters([
142         ('s1': [1, 2, 3],
143          's2': [CT(['a', 'b']), 'c', 'd'],
144          'expand_composites': False,
145          'expected': [CT(['a', 'b']), 'c', 'd'],
146          'paths': [(0,), (1,), (2,)]],
147
148         ('expand_composites': raise,
149          'expected': [CT(['a', 'b']), 'c', 'd'],
150          'paths': [(0,), (1,), (2,)]],
151         ('s1': [CT([1, 2, 3]),
152          's2': [CT([1, 2, 3]),
153          'expand_composites': False,
154          'paths': [(0, 0), (1, 1), (2, 'y')]],
155          's1': [CT([9, 9, 9]), 999, ('y': CT([9, 9])),
156          's2': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
157          'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
158          'paths': [(0, 0), (1, 1), (2, 'y')]],
159          's1': [CT([9, 9, 9]), 999, ('y': CT([CT([9, 9]), 9)]),
160          's2': [CT([1, 2, 3]), 100, ('y': CT([5, 6])),
161          'expand_composites': False,
162          'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
163          'paths': [(0, 0), (1, 1), (2, 'y')]],
164         ]) # pyformat: disable
165     def testNestFlattenUpTo(self, s1, s2, expected, paths,
166                             expand_composites=True):
167         result = nest.flatten_up_to(s1, s2, expand_composites=expand
168         self.assertEqual(result, expected)
169
170         result_with_paths = nest.flatten_with_tuple_paths_up_to(
171             s1, s2, expand_composites=expand_composites)
172         self.assertEqual(result_with_paths, list(zip(paths, expected
173
174     @parameterized.parameters([
175         ('structure': CT(0),
176          'sequence': [5],
177          'expected': CT(5)),
178         ('structure': CT(['a', 'b', 'c']),
179          'sequence': ['a', CT(['b']), 'x': 'y']),
180          'expected': [CT(['a', CT(['b']), 'x': 'y']),
181          ('structure': [['k1': CT('a')], CT(['b', 'x': CT('y')
182              'sequence': ['a', 'b', 'c'],
183              'expected': [['k1': CT('a')], CT(['b', 'x': CT('y')
184              ('structure': [['k1': CT('a')], CT(['b', 'x': CT('y')
185              'sequence': ['a', 'b'],
186              'expand_composites': False,
187              'expected': [5],
188              'sequence': [CT(5, metadata='abc')],
189              'expected': CT(5, metadata='abc')],
190         ]) # pyformat: disable
191     def testNestPackSequence(self,
192                             structure,
193                             sequence,
194                             expected,
195                             expand_composites=True):
196         result = nest.pack_sequence_and(
197             structure, sequence, expand_composites=expand_composites
198         self.assertEqual(result, expected)
199
200     @parameterized.parameters([
201         ('s1': CT('abc'), 's2': CT('xyz')),
202         ('s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f']),
203         ('s1': [1, CT([10]), CT([200, metadata='xyz']),
204          's2': [8, CT([95]), CT([100, metadata='xyz'])],
205         ]) # pyformat: disable
206     def testNestAssertSameStructure(self, s1, s2, expand_composit
207         nest.assert_same_structure(s1, s2, expand_composites=expand
208         nest.assert_shallow_structure(s1, s2, expand_composites=expa
209
210     @parameterized.parameters([
211         ('s1': CT(0), 's2': CT(['x'])),
212         ('s1': CT([1]), 's2': CT([1, 2])),
213         ('s1': CT(['a', 'b', 'c']), 's2': CT(['y': 3]),
214         ('s1': CT(0), 's2': CT(0, metadata='xyz')),
215         ('s1': CT(0, metadata='xyz'), 's2': CT(0)),
216         ('s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')],
217         ('s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])),
218         ('s1': [1, CT(['a']), CT('b', metadata='xyz')],
219
220         ('s2': [8, CT([95, 66]), CT([100, metadata='abc'])],
221         ('s1': CT(0), 's2': CT(0), 'error': TypeError),
222         ]) # pyformat: disable
223     def testNestAssertSameStructureComposeMismatch(self,
224                                                     s1,
225                                                     s2,
226                                                     error=ValueError):
227         # s1 and s2 have the same structure if expand_composites=False; but
228         # different structures if expand_composites=True.
229         nest.assert_same_structure(s1, s2, expand_composites=False)
230         nest.assert_shallow_structure(s1, s2, expand_composites=False)
231         with self.assertRaises(error): # pylint: disable=error-prone-assert-raises
232             nest.assert_same_structure(s1, s2, expand_composites=True)
233
234     @parameterized.parameters([
235         # Note: there are additional test cases in testNestAssertSameStructure.
236         ('s1': [1], 's2': [CT([1])],
237          's1': [CT([1, 2, 3]), 100, ('y': CT([5, 6])],
238          's2': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6])],
239          'expand_composites': False,
240          ('s1': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6])],
241           's2': [CT([1, 2, 3]), 100, ('y': CT([5, 6])],
242           'expand_composites': False,
243         ]) # pyformat: disable
244     def testNestAssertShallowStructure(self, s1, s2, expand_composites=True):
245         nest.assert_shallow_structure(s1, s2, expand_composites=expand_composit
246
247     @parameterized.parameters([
248         # Note: there are additional test cases in
249         # testNestAssertSameStructureComposeMismatch.
250         ('s1': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6])],
251          's2': [CT([1, 2, 3]), 100, ('y': CT([5, 6])],
252          ('s1': CT([1, 2, 3]),
253           's2': [1, 2, 3],
254           'check_types': False,
255         ]) # pyformat: disable
256     def testNestAssertShallowStructureComposeMismatch(self,
257                                                     s1,
258                                                     s2,
259                                                     check_types=True):
260         with self.assertRaises((TypeError, ValueError)): # pylint: disable=g-error-prone-assert-raises
261             nest.assert_shallow_structure(
262                 s1, s2, expand_composites=True, check_types=check_types)
263
264     @parameterized.parameters([
265         ('structure': CT(1, metadata=2),
266          'expected': CT([1, metadata=2]),
267         ('structure': CT(['x': 1, 'y': [2, 3]], metadata=2),
268          'expected': CT(['x': 1, 'y': [12, 13]], metadata=2),
269         ('structure': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6])],
270          'expected': [CT([11, 12, 13]), 110, ('y': CT([CT([14, 15]), 16])],
271         ]) # pyformat: disable
272     def testNestMapStructure(self, structure, expected, expand_composites=True):
273         func = lambda x: x + 10
274         result = nest.map_structure(
275             func, structure, expand_composites=expand_composites)
276         self.assertEqual(result, expected)
277
278     @parameterized.parameters([
279         ('s1': [CT([1, 2, 3]), 100, ('y': 4)],
280          's2': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6])],
281          'expected': [CT([11, 12, 13]), 110, ('y': CT([CT([4, 5]), 6])],
282         ]) # pyformat: disable
283     def testNestMapStructureViaDo(self, s1, s2, expected):
284         func = lambda x: x + 10 if isinstance(x, int) else x
285         result = nest.map_structure_up_to(s1, func, s2, expand_composites=True)
286         self.assertEqual(result, expected)
287
288     @parameterized.parameters([
289         ('structure': CT('a'),
290          'expected': CT(['CT:a']),
291         ('structure': CT(['a', 'b']),
292          'expected': CT(['CT:a/a', 'CT:b/b']),
293

```

```

76 def _repr_(self):
77     return '%s(%r, %r)' % (type(self).__name__, self.comp
78
79 def __eq__(self, other):
80     return (type(self) is type(other) and
81             self.components == other.components and
82             self.metadata == other.metadata)
83
84 # Another test case for Tensor class. 'tf.nest' should
85 # be able to handle different types (e.g. for assert_
86 class CTSpec(CTSpec):
87     pass
88
89 class CT2(CT):
90     _type_spec_class = CTSpec
91
92 @test_util.run_all_in_graph_and_eager_modes
93 class CompositeTensorTest(test_util.TensorFlowTestCase,
94                             @parameterized.parameters([
95     ('structure': CT(0),
96      'expected': [0],
97      'paths': [['CT']],
98      'structure': CT('a'),
99      'expected': ['a'],
100      'paths': ['CT'],
101      'structure': CT('a', 'b', 'c'),
102      'expected': ['a', 'b', 'c'],
103      'paths': ['CT', 'a'],
104      'structure': CT('a', 'b', 'c'),
105      'expected': ['a', 'b', 'c'],
106      'paths': ['CT', 'a'],
107      'structure': CT('a', 'b', 'c'),
108      'expected': ['a', 'b', 'c'],
109      'paths': ['CT', 'a'],
110      'structure': CT('a', 'b', 'c'),
111      'expected': ['a', 'b', 'c'],
112      'paths': ['CT', 'a'],
113      'structure': CT('a', 'b', 'c'),
114      'expected': ['a', 'b', 'c'],
115      'paths': ['CT', 'a'],
116      'structure': CT('a', 'b', 'c'),
117      'expected': ['a', 'b', 'c'],
118      'paths': ['CT', 'a'],
119      'structure': CT('a', 'b', 'c'),
120      'expected': ['a', 'b', 'c'],
121      'paths': ['CT', 'a'],
122      'structure': CT('a', 'b', 'c'),
123      'expected': ['a', 'b', 'c'],
124      'paths': ['CT', 'a'],
125      'structure': CT('a', 'b', 'c'),
126      'expected': ['a', 'b', 'c'],
127      'paths': ['CT', 'a'],
128      'structure': CT('a', 'b', 'c'),
129      'expected': ['a', 'b', 'c'],
130      'paths': ['CT', 'a'],
131      'structure': CT('a', 'b', 'c'),
132      'expected': ['a', 'b', 'c'],
133      'paths': ['CT', 'a'],
134      'structure': CT('a', 'b', 'c'),
135      'expected': ['a', 'b', 'c'],
136      'paths': ['CT', 'a'],
137      'structure': CT('a', 'b', 'c'),
138      'expected': ['a', 'b', 'c'],
139      'paths': ['CT', 'a'],
140      'structure': CT('a', 'b', 'c'),
141      'expected': ['a', 'b', 'c'],
142      'paths': ['CT', 'a'],
143      'structure': CT('a', 'b', 'c'),
144      'expected': ['a', 'b', 'c'],
145      'paths': ['CT', 'a'],
146      'structure': CT('a', 'b', 'c'),
147      'expected': ['a', 'b', 'c'],
148      'paths': ['CT', 'a'],
149      'structure': CT('a', 'b', 'c'),
150      'expected': ['a', 'b', 'c'],
151      'paths': ['CT', 'a'],
152      'structure': CT('a', 'b', 'c'),
153      'expected': ['a', 'b', 'c'],
154      'paths': ['CT', 'a'],
155      'structure': CT('a', 'b', 'c'),
156      'expected': ['a', 'b', 'c'],
157      'paths': ['CT', 'a'],
158      'structure': CT('a', 'b', 'c'),
159      'expected': ['a', 'b', 'c'],
160      'paths': ['CT', 'a'],
161      'structure': CT('a', 'b', 'c'),
162      'expected': ['a', 'b', 'c'],
163      'paths': ['CT', 'a'],
164      'structure': CT('a', 'b', 'c'),
165      'expected': ['a', 'b', 'c'],
166      'paths': ['CT', 'a'],
167      'structure': CT('a', 'b', 'c'),
168      'expected': ['a', 'b', 'c'],
169      'paths': ['CT', 'a'],
170      'structure': CT('a', 'b', 'c'),
171      'expected': ['a', 'b', 'c'],
172      'paths': ['CT', 'a'],
173      'structure': CT('a', 'b', 'c'),
174      'expected': ['a', 'b', 'c'],
175      'paths': ['CT', 'a'],
176      'structure': CT('a', 'b', 'c'),
177      'expected': ['a', 'b', 'c'],
178      'paths': ['CT', 'a'],
179      'structure': CT('a', 'b', 'c'),
180      'expected': ['a', 'b', 'c'],
181      'paths': ['CT', 'a'],
182      'structure': CT('a', 'b', 'c'),
183      'expected': ['a', 'b', 'c'],
184      'paths': ['CT', 'a'],
185      'structure': CT('a', 'b', 'c'),
186      'expected': ['a', 'b', 'c'],
187      'paths': ['CT', 'a'],
188      'structure': CT('a', 'b', 'c'),
189      'expected': ['a', 'b', 'c'],
190      'paths': ['CT', 'a'],
191      'structure': CT('a', 'b', 'c'),
192      'expected': ['a', 'b', 'c'],
193      'paths': ['CT', 'a'],
194      'structure': CT('a', 'b', 'c'),
195      'expected': ['a', 'b', 'c'],
196      'paths': ['CT', 'a'],
197      'structure': CT('a', 'b', 'c'),
198      'expected': ['a', 'b', 'c'],
199      'paths': ['CT', 'a'],
200      'structure': CT('a', 'b', 'c'),
201      'expected': ['a', 'b', 'c'],
202      'paths': ['CT', 'a'],
203      'structure': CT('a', 'b', 'c'),
204      'expected': ['a', 'b', 'c'],
205      'paths': ['CT', 'a'],
206      'structure': CT('a', 'b', 'c'),
207      'expected': ['a', 'b', 'c'],
208      'paths': ['CT', 'a'],
209      'structure': CT('a', 'b', 'c'),
210      'expected': ['a', 'b', 'c'],
211      'paths': ['CT', 'a'],
212      'structure': CT('a', 'b', 'c'),
213      'expected': ['a', 'b', 'c'],
214      'paths': ['CT', 'a'],
215      'structure': CT('a', 'b', 'c'),
216      'expected': ['a', 'b', 'c'],
217      'paths': ['CT', 'a'],
218      'structure': CT('a', 'b', 'c'),
219      'expected': ['a', 'b', 'c'],
220      'paths': ['CT', 'a'],
221      'structure': CT('a', 'b', 'c'),
222      'expected': ['a', 'b', 'c'],
223      'paths': ['CT', 'a'],
224      'structure': CT('a', 'b', 'c'),
225      'expected': ['a', 'b', 'c'],
226      'paths': ['CT', 'a'],
227      'structure': CT('a', 'b', 'c'),
228      'expected': ['a', 'b', 'c'],
229      'paths': ['CT', 'a'],
230      'structure': CT('a', 'b', 'c'),
231      'expected': ['a', 'b', 'c'],
232      'paths': ['CT', 'a'],
233      'structure': CT('a', 'b', 'c'),
234      'expected': ['a', 'b', 'c'],
235      'paths': ['CT', 'a'],
236      'structure': CT('a', 'b', 'c'),
237      'expected': ['a', 'b', 'c'],
238      'paths': ['CT', 'a'],
239      'structure': CT('a', 'b', 'c'),
240      'expected': ['a', 'b', 'c'],
241      'paths': ['CT', 'a'],
242      'structure': CT('a', 'b', 'c'),
243      'expected': ['a', 'b', 'c'],
244      'paths': ['CT', 'a'],
245      'structure': CT('a', 'b', 'c'),
246      'expected': ['a', 'b', 'c'],
247      'paths': ['CT', 'a'],
248      'structure': CT('a', 'b', 'c'),
249      'expected': ['a', 'b', 'c'],
250      'paths': ['CT', 'a'],
251      'structure': CT('a', 'b', 'c'),
252      'expected': ['a', 'b', 'c'],
253      'paths': ['CT', 'a'],
254      'structure': CT('a', 'b', 'c'),
255      'expected': ['a', 'b', 'c'],
256      'paths': ['CT', 'a'],
257      'structure': CT('a', 'b', 'c'),
258      'expected': ['a', 'b', 'c'],
259      'paths': ['CT', 'a'],
260      'structure': CT('a', 'b', 'c'),
261      'expected': ['a', 'b', 'c'],
262      'paths': ['CT', 'a'],
263      'structure': CT('a', 'b', 'c'),
264      'expected': ['a', 'b', 'c'],
265      'paths': ['CT', 'a'],
266      'structure': CT('a', 'b', 'c'),
267      'expected': ['a', 'b', 'c'],
268      'paths': ['CT', 'a'],
269      'structure': CT('a', 'b', 'c'),
270      'expected': ['a', 'b', 'c'],
271      'paths': ['CT', 'a'],
272      'structure': CT('a', 'b', 'c'),
273      'expected': ['a', 'b', 'c'],
274      'paths': ['CT', 'a'],
275      'structure': CT('a', 'b', 'c'),
276      'expected': ['a', 'b', 'c'],
277      'paths': ['CT', 'a'],
278      'structure': CT('a', 'b', 'c'),
279      'expected': ['a', 'b', 'c'],
280      'paths': ['CT', 'a'],
281      'structure': CT('a', 'b', 'c'),
282      'expected': ['a', 'b', 'c'],
283      'paths': ['CT', 'a'],
284      'structure': CT('a', 'b', 'c'),
285      'expected': ['a', 'b', 'c'],
286      'paths': ['CT', 'a'],
287      'structure': CT('a', 'b', 'c'),
288      'expected': ['a', 'b', 'c'],
289      'paths': ['CT', 'a'],
290      'structure': CT('a', 'b', 'c'),
291      'expected': ['a', 'b', 'c'],
292      'paths': ['CT', 'a'],
293      'structure': CT('a', 'b', 'c'),
294      'expected': ['a', 'b', 'c'],
295      'paths': ['CT', 'a'],
296      'structure': CT('a', 'b', 'c'),
297      'expected': ['a', 'b', 'c'],
298      'paths': ['CT', 'a'],
299      'structure': CT('a', 'b', 'c'),
300      'expected': ['a', 'b', 'c'],
301      'paths': ['CT', 'a'],
302      'structure': CT('a', 'b', 'c'),
303      'expected': ['a', 'b', 'c'],
304      'paths': ['CT', 'a'],
305      'structure': CT('a', 'b', 'c'),
306      'expected': ['a', 'b', 'c'],
307      'paths': ['CT', 'a'],
308      'structure': CT('a', 'b', 'c'),
309      'expected': ['a', 'b', 'c'],
310      'paths': ['CT', 'a'],
311      'structure': CT('a', 'b', 'c'),
312      'expected': ['a', 'b', 'c'],
313      'paths': ['CT', 'a'],
314      'structure': CT('a', 'b', 'c'),
315      'expected': ['a', 'b', 'c'],
316      'paths': ['CT', 'a'],
317      'structure': CT('a', 'b', 'c'),
318      'expected': ['a', 'b', 'c'],
319      'paths': ['CT', 'a'],
320      'structure': CT('a', 'b', 'c'),
321      'expected': ['a', 'b', 'c'],
322      'paths': ['CT', 'a'],
323      'structure': CT('a', 'b', 'c'),
324      'expected': ['a', 'b', 'c'],
325      'paths': ['CT', 'a'],
326      'structure': CT('a', 'b', 'c'),
327      'expected': ['a', 'b', 'c'],
328      'paths': ['CT', 'a'],
329      'structure': CT('a', 'b', 'c'),
330      'expected': ['a', 'b', 'c'],
331      'paths': ['CT', 'a'],
332      'structure': CT('a', 'b', 'c'),
333      'expected': ['a', 'b', 'c'],
334      'paths': ['CT', 'a'],
335      'structure': CT('a', 'b', '
```



```

74     return self._type_spec.class(component_specs, self.meta
75
76     def __repr__(self):
77         return '%s(%r, %r)' % (type(self).__name__, self.comp
78
79     def __eq__(self, other):
80         return (type(self) is type(other) and
81                 self.components == other.components and
82                 self.metadata == other.metadata)
83
84
85 # Another test CompositeTensor class. 'tf.nest' should h
86 # classes as different structure types (e.g. for assert_s
87 class CTSpec2(CTSpec):
88     pass
89
90
91 class CT2(CT):
92     _type_spec.class = CTSpec2
93
94
95 @test_util.run_all_in_graph_and_eager_modes
96 def testTensorFlowTest(test_util.TensorFlowTestCase, p
97
98     @parameterized.parameters([
99         ('structure': CT(0)),
100         ('expected': [0]),
101         ('paths': ['CT(''])),
102         ('structure': CT('a')),
103         ('expected': ['a']),
104         ('paths': ['CT(''])),
105         ('structure': CT(['a', 'b', 'c']),
106         ('expected': ['a', 'b', 'c']),
107         ('paths': ['CT('', 0], ('CT', 1], ('CT', 2)])),
108         ('structure': CT(['x': 'a', 'y': 'b', 'z': 'c']),
109         ('expected': ['a', 'b', 'c']),
110         ('paths': ['CT('', 'x'], ('CT', 'y'], ('CT', 'z')])),
111         ('structure': CT(['k1': 'CT('a', 'CT('b', 'x': CT(
112         ('expected': ['a', 'b', 'c']),
113         ('paths': ['0, 'k1', 'CT', (1, 'CT', 0), (1, 'CT',
114         ('structure': CT(0)),
115         ('expected_composites': False),
116         ('expected': [CT(0)],
117         ('paths': [()])),
118         ('structure': CT(['k1': CT('a')], CT('b', 'x': CT(
119         ('expected_composites': False),
120         ('expected': [CT('a'), CT('b', 'x': CT('y': 'c')
121         ('paths': ['0, 'k1', (1, 1)])),
122     ]) # pyformat: disable
123 def testNestFlatten(self, structure, expected, paths, e
124     result = nest.flatten(structure, expand_composites=ex
125     self.assertEqual(result, expected)
126
127     result_with_paths = nest.flatten_with_tuple_paths(
128         structure, expand_composites=expand_composites)
129     self.assertEqual(result_with_paths, list(zip(paths, e
130
131     string_paths = ['/'.join(str(p) for p in path) for pa
132     result_with_string_paths = nest.flatten_with_joined_s
133         structure, expand_composites=expand_composites)
134     self.assertEqual(result_with_string_paths,
135         list(zip(string_paths, expected)))
136
137     flat_paths_result = list(
138         nest_yield_flat_paths(structure, expand_composit
139     self.assertEqual(flat_paths_result, paths)
140
141     @parameterized.parameters([
142         ('s1': [1, 2, 3]),
143         ('s2': [CT(['a', 'b']), 'c', 'd']),
144         ('expand_composites': False),
145         ('expected': [CT(['a', 'b']), 'c', 'd']),
146         ('paths': [(0, 1), (1, 2, 3)]),
147
148     expand_composites = raise,
149     'expected': [CT(['a', 'b']), 'c', 'd']),
150     'paths': [(0, 1), (1, 2, 3)]),
151     's2': [CT([1, 2, 3])],
152     'expand_composites': False,
153     'paths': [(0, 1), (1, 2, 3)]),
154     's1': [CT([9, 9, 9]), 999, ('y': CT([9, 9]))],
155     's2': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6]))],
156     'expand_composites': False,
157     'paths': [(0, 1), (1, 2, 3)], 100, CT([CT([4, 5]), 6])),
158     's1': [CT([9, 9, 9]), 999, ('y': CT([CT([9, 9]), 9]))],
159     's2': [CT([1, 2, 3]), 100, ('y': CT([5, 6]))],
160     'expand_composites': False,
161     'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
162     'paths': [(0, 0), (1, 2, 'y')]),
163     ]) # pyformat: disable
164 def testNestFlattenTo(self, s1, s2, expected, paths,
165     expand_composites=True):
166     result = nest.flatten_to(s1, s2, expand_composites=expa
167     result_with_paths = nest.flatten_with_tuple_paths_to(
168         s1, s2, expand_composites=expand_composites)
169     self.assertEqual(result_with_paths, list(zip(paths, expecte
170
171     @parameterized.parameters([
172         ('structure': CT(0)),
173         ('sequence': [5]),
174         ('expected': CT(5)),
175         ('structure': CT(['a', 'b', 'c']),
176         ('sequence': ['a', CT(['b']), 'x': 'y']),
177         ('expected': [CT('a', CT(['b']), 'x': 'y')]),
178         ('structure': [CT('a': CT('a')), CT('b', 'x': CT('y'
179         ('sequence': ['a', 'b', 'c']),
180         ('expected': [CT('a'), CT('b', 'x': CT('y'
181         ('structure': [CT('k1': CT('a')), CT('b', 'x': CT('y'
182         ('sequence': ['a', 'b']),
183         ('expand_composites': False),
184         ('expected': [CT('k1': 'a', 'b')]),
185         ('structure': CT(0, metadata='abc')),
186         ('sequence': [5]),
187         ('expected': CT(5, metadata='abc')),
188     ]) # pyformat: disable
189 def testNestPackSequences(self,
190     structure,
191     sequence,
192     expected,
193     expand_composites=True):
194     result = nest.pack_sequence_as(
195         structure, sequence, expand_composites=expand_composit
196     self.assertEqual(result, expected)
197
198     @parameterized.parameters([
199         ('s1': CT('abc'), 's2': CT('xyz')),
200         ('s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])),
201         ('s1': [1, CT([10]), CT([200, metadata='xyz']),
202         ('s2': [8, CT([55]), CT([100, metadata='xyz')]),
203     ]) # pyformat: disable
204 def testNestAssertSameStructure(self, s1, s2, expand_composit
205     nest.assert_same_structure(s1, s2, expand_composites=expa
206     nest.assert_shallow_structure(s1, s2, expand_composites=ex
207
208     @parameterized.parameters([
209         ('s1': CT(0), 's2': CT('x')),
210         ('s1': CT([1]), 's2': CT([1, 2])),
211         ('s1': CT(['a', 'b']), 's2': CT(['y': 3])),
212         ('s1': CT(0, metadata='xyz'), 's2': CT(0)),
213         ('s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')
214         ('s1': CT(['a', 'b', 'c']), 's2': CT('d', 'e')),
215         ('s1': CT(['a', 'b', 'c'], CT('a', 'b', metadata='xyz')
216
217     's2': [8, CT([55, 66]), CT([100, metadata='abc'])],
218     ('s1': CT(0), 's2': CT(0), 'error': TypeError),
219     ]) # pyformat: disable
220 def testNestAssertSameStructureCompositeMismatch(self,
221     s1,
222     s2,
223     error=ValueError),
224     # s1 and s2 have the same structure if expand_composites=Tru
225     # different structures if expand_composites=False
226     nest.assert_same_structure(s1, s2, expand_composites=False)
227     nest.assert_shallow_structure(s1, s2, expand_composites=Fa
228     with self.assertRaises(error): # pylint: disable=error-pr
229         nest.assert_same_structure(s1, s2, expand_composites=Tru
230
231     @parameterized.parameters([
232         # Note: there are additional test cases in testNestAssert
233         ('s1': [1], 's2': [CT([3])],
234         ('s1': [CT([1, 2, 3])], 100, ('y': CT([5, 6]))],
235         ('s1': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6]))]
236         'expand_composites': False,
237         ('s1': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6]))]
238         's2': [CT([1, 2, 3]), 100, ('y': CT([5, 6]))],
239         'expand_composites': False,
240     def testNestAssertShallStructure(self, s1, s2, expand_com
241     nest.assert_shallow_structure(s1, s2, expand_composites=exp
242
243     @parameterized.parameters([
244         # Note: there are additional test cases in
245         ('structure': CT(['a', 'b', 'c']),
246         ('sequence': ['a', CT(['b']), 'x': 'y']),
247         ('expected': [CT('a', CT(['b']), 'x': 'y')]),
248         ('structure': [CT('a': CT('a')), CT('b', 'x': CT('y'
249         ('sequence': ['a', 'b', 'c']),
250         ('expected': [CT('a'), CT('b', 'x': CT('y'
251         ('structure': [CT('k1': CT('a')), CT('b', 'x': CT('y'
252         ('sequence': ['a', 'b']),
253         ('expand_composites': False),
254         ('expected': [CT('k1': 'a', 'b')]),
255         ('structure': CT(0, metadata='abc')),
256         ('sequence': [5]),
257         ('expected': CT(5, metadata='abc')),
258     ]) # pyformat: disable
259 def testNestPackSequences(self,
260     structure,
261     sequence,
262     expected,
263     expand_composites=True):
264     result = nest.pack_sequence_as(
265         structure, sequence, expand_composites=expand_composit
266     self.assertEqual(result, expected)
267
268     @parameterized.parameters([
269         ('s1': CT('abc'), 's2': CT('xyz')),
270         ('s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])),
271         ('s1': [1, CT([10]), CT([200, metadata='xyz']),
272         ('s2': [8, CT([55]), CT([100, metadata='xyz')]),
273     ]) # pyformat: disable
274 def testNestMapStructure(self, structure, expected, expand_co
275     func = lambda x: x + 10
276     self.assertEqual(result, expected)
277
278     @parameterized.parameters([
279         ('s1': [CT([1, 2, 3])], 100, ('y': 4]),
280         ('s2': [CT([1, 2, 3]), 100, ('y': CT([CT([4, 5]), 6]))]
281         'expected': [[CT([1, 12, 13]), 110, ('y': CT([CT([4, 5
282     ]) # pyformat: disable
283 def testAssertSameStructure(self, self, s1, s2, expected):
284     func = lambda x: x + 10 if isinstance(x, int) else x
285     result = nest.map_structure_to(s1, func, s2, expand_com
286     self.assertEqual(result, expected)
287
288     @parameterized.parameters([
289         ('structure': CT('a')),
290         ('expected': CT('CT:a')),
291         ('structure': CT(['a', 'b']),
292         ('expected': CT(['CT:a', 'CT:b'])),
293
294     'Result' = nest.map_structure(
295         func, structure, expand_composites=expand_composites)
296     self.assertEqual(result, expected)
297
298     @parameterized.parameters([
299         ('structure': CT('a')),
300         ('expected': CT('CT:a')),
301         ('structure': CT(['a', 'b']),
302         ('expected': CT(['CT:a', 'CT:b'])),
303         ('structure': CT(['CT:a', 'CT:b']),
304         ('expected': CT(['CT:a', 'CT:b'])),
305     ]) # pyformat: disable
306 def testNestMapStructureTo(self, self, s1, s2, expected):
307     func = lambda x: x + 10 if isinstance(x, int) else x
308     result = nest.map_structure_to(s1, func, s2, expand_composit
309     self.assertEqual(result, expected)
310
311     @parameterized.parameters([
312         ('structure': CT('a')),
313         ('expected': CT('CT:a')),
314         ('structure': CT(['a', 'b']),
315         ('expected': CT(['CT:a', 'CT:b'])),
316         ('structure': CT(['CT:a', 'CT:b']),
317         ('expected': CT(['CT:a', 'CT:b'])),
318     ]) # pyformat: disable
319 def testAssertSameStructureWithValueAndTypeSpec(self, value_func):
320     value = value_func()
321     spec = nest.map_structure(type_spec.type_spec_from_value, value,
322     expand_composites=False)
323     nest.assert_same_structure(value, spec, expand_composites=True)
324
325     if __name__ == '__main__':
326         googletest.main()

```



Machine Learning

- Get data
- Define & calculate features
- Train and use the model

Getting data



SPAM

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

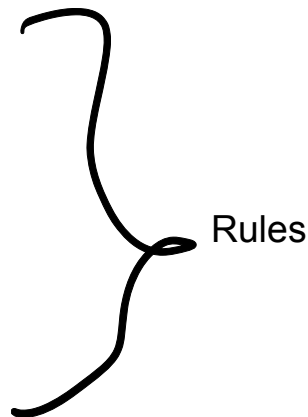
Congratulations again!

Machine Learning

- Get data
- **Define & calculate features**
- Train and use the model

Features

- Length of title > 10? true/false
- Length of body > 10? true/false
- Sender "promotions@online.com"? true/false
- Sender "hpYOSKml@test.com"? true/false
- Sender domain "test.com"? true/false
- Description contains "deposit"? true/false





Start with rules and then use these rules
as features

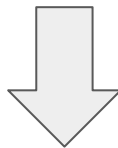
Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM



[1, 1, 0, 0, 1, 1]

Subject: Waiting for your reply


From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

Length of title > 10? **True**


[1, 1, 0, 0, 1, 1]

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

[1, 1, 0, 0, 1, 1]

Length of body > 10? **True**

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

Sender "[promotions@online.com](#)"? **False**

[1, 1, 0, 0, 1, 1]

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

[1, 1, 0, 0, 1, 1]

Sender "[hpYOSKmL@test.com](#)"? **False**

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

Sender domain "test.com"? **True**

[1, 1, 0, 0, 1, 1]

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please **deposit** \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

[1, 1, 0, 0, 1, **1**]

TRUE

Description contains "deposit"? ~~False~~

Subject: Waiting for your reply

From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit \$10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

SPAM

[1, 1, 0, 0, 1, 1]

1

Features
(data)

Target
(desired output)

[1, 1, 0, 0, 1, 1]

1

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1

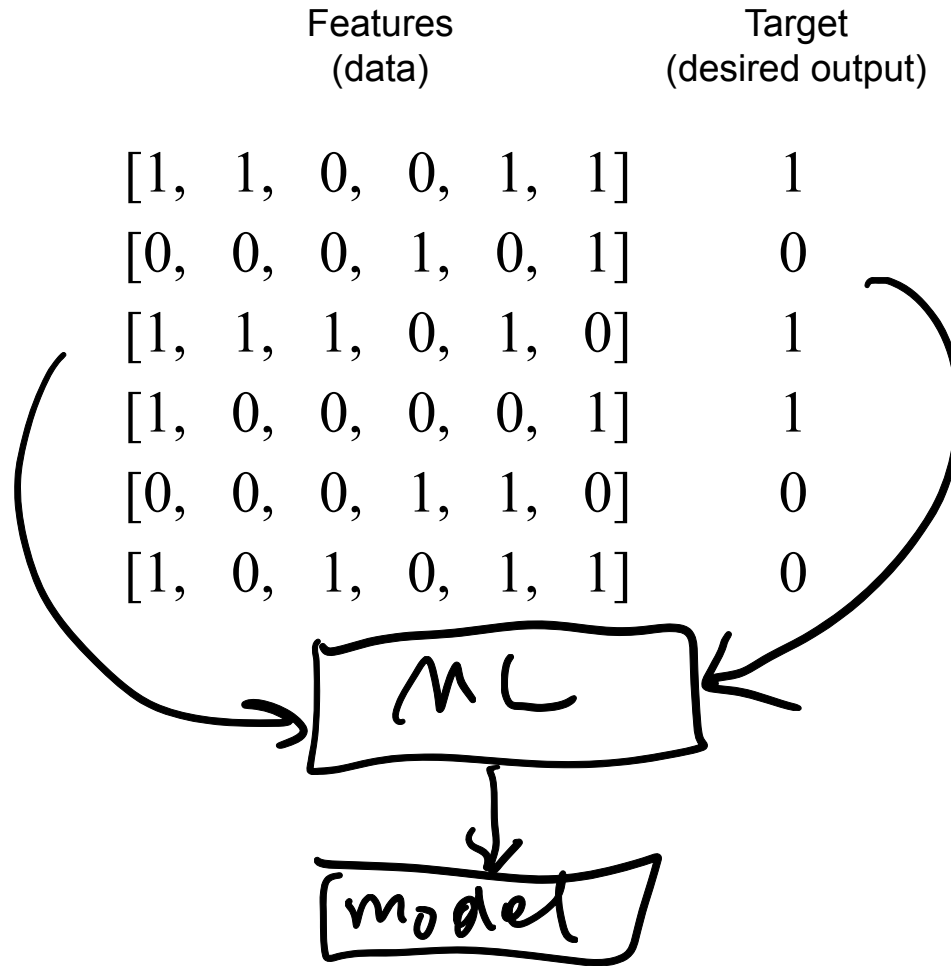
Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 1]	0

Machine Learning

- Get data
- Define & calculate features
- **Train and use the model**

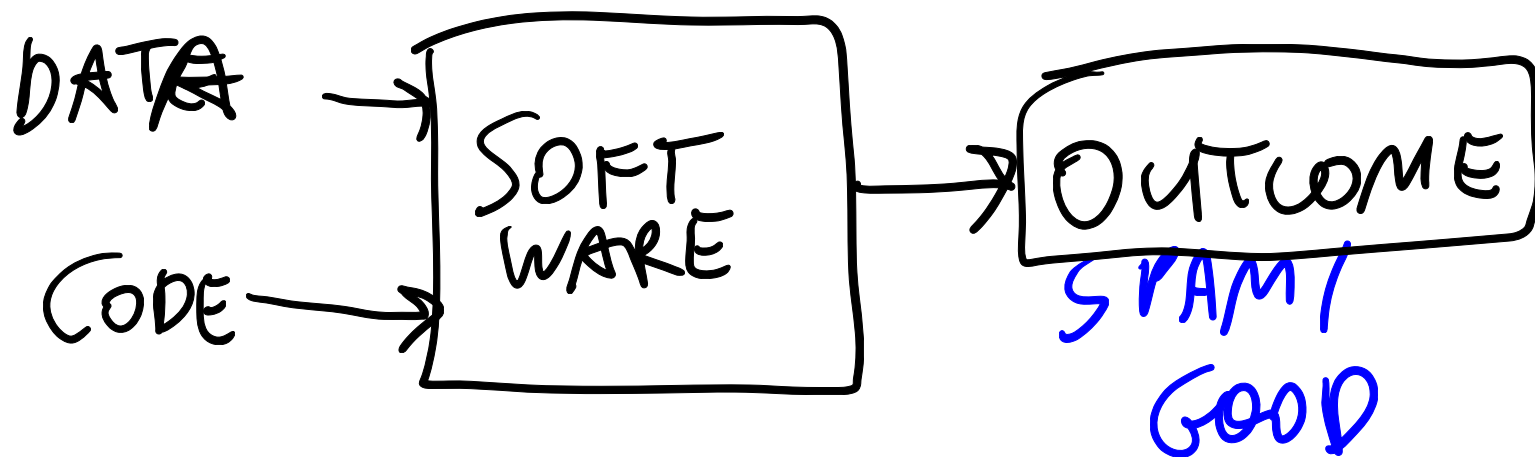


Apply

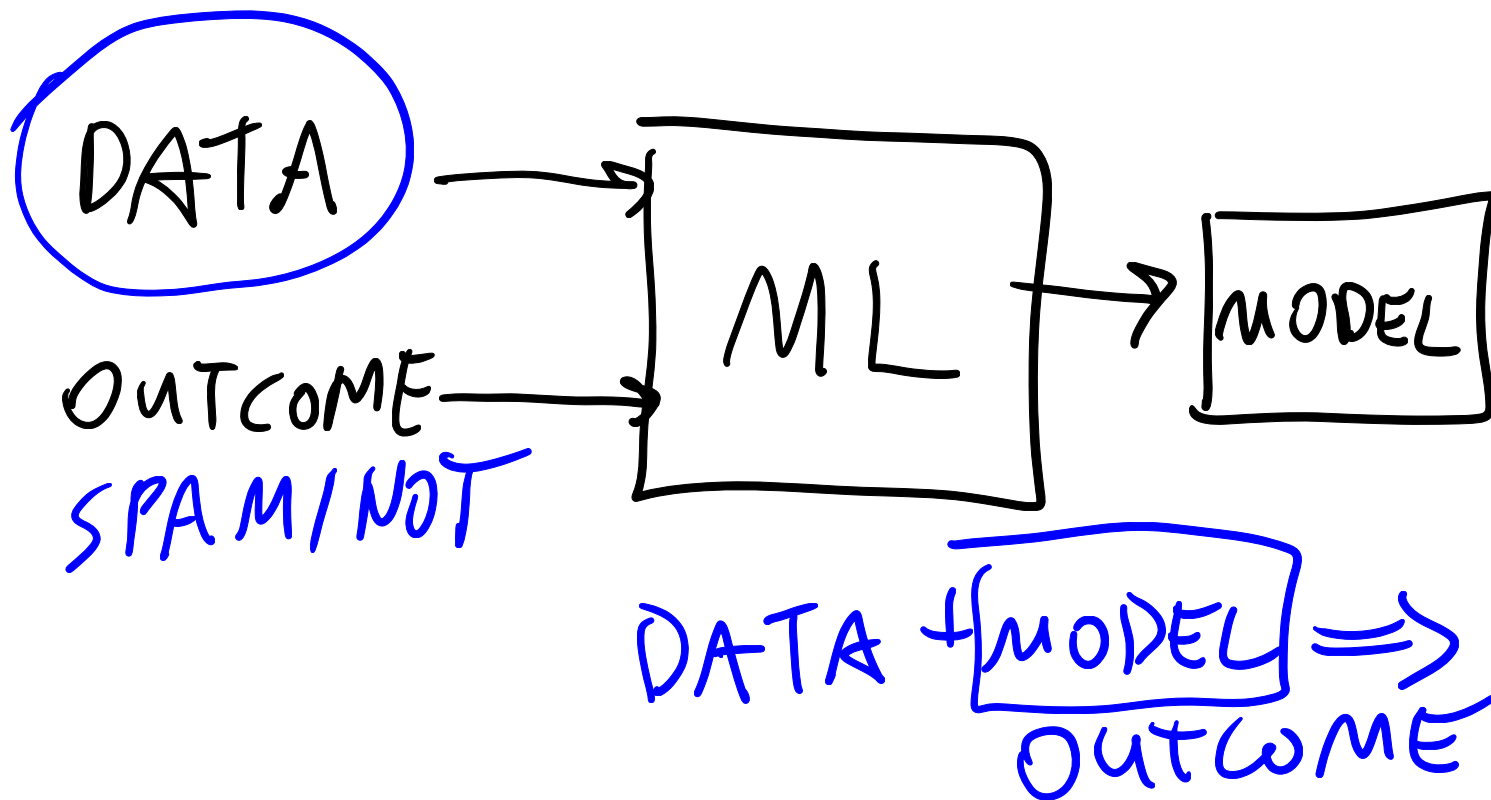
Model

	Features (data)	Predictions (output)	Final outcome (decision)
→	[0, 0, 0, 1, 0, 1]	0.8	SPAM
	[0, 0, 0, 1, 1, 0]	0.6	S
	[1, 0, 1, 0, 1, 1]	0.1	GOOD
	[1, 1, 1, 0, 1, 0]	0.21	G
	[1, 0, 0, 0, 0, 1]	0.7	S
	[1, 1, 0, 0, 1, 1]	0.4	G

 ≥ 0.5



data + code => software => outcome



data + outcome => ML => model

Next

Supervised machine learning

- A bit more formal definition
- Examples: regression, classification, ranking