```
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)

def create_model(**kwargs):
  """Create model from global arguments."""
  # Load in the model
  model = build_model(MODEL_NAME, MODEL_WF,
                      char_size=len(CHAR_IDX)+1,
                      dim=ARGS.dim,
                      **kwargs)
  if ARGS.summary:
    model.summary()
  return model

def evaluate():
  """Evaluate model on each test data."""
  model = create_model(iterations=ARGS.iterations, training=True)
  # training, run, glove, pararule, num = MODEL_FNAME.split('_')
  #training, run, glove, pararule = MODEL_FNAME.split('_')
  name_list = []
  name_list = MODEL_FNAME.split('_')

  for s in ['val_task0.jsonl','val_task1.jsonl']:
            #for      s      in      ['test_depth_1.jsonl',      'test_depth_2.jsonl',      'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
  # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
  #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
    results = list()
    dgen = LogicSeq.from_file("data/ConceptRules/dev/"+s, ARGS.batch_size, pad=ARGS.pad, verbose=False)
    _, acc = model.evaluate_generator(dgen) # [loss, acc]
    results.append(acc)
    print(name_list[0], ARGS.model, ARGS.dim, s, name_list[1], acc, sep=',')
  #print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')

def showsave_plot():
  """Show or save plot."""
  if ARGS.outf:
    plt.savefig(ARGS.outf, bbox_inches='tight')
  else:
    plt.show()

def eval_nstep():
  """Evaluate model on nstep deduction."""
  # Evaluate model on every nstep test data
  results = list()
  for i in range(1, 33):
    dgen = LogicSeq.from_file("data/test_nstep{}.txt".format(i), ARGS.batch_size, pad=ARGS.pad, verbose=False)
    model = create_model(iterations=max(ARGS.iterations, i+1), training=True)
```

```
    results.append(model.evaluate_generator(dgen)[1])
  training, _, run = MODEL_FNAME.split('_')
  print(training, ARGS.model, ARGS.dim, run, *results, sep=',')

def plot_nstep():
  """Plot nstep results."""
  # Plot the results
  df = pd.read_csv("nstep_results.csv")
  # Get maximum run based on mean
  df['Mean'] = df.iloc[:,4:].mean(axis=1)
  idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
  df = df.loc[idx]
  df = df.drop(columns=['Mean'])
  df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
  df['NStep'] = df['NStep'].astype(int)
  df = df[(df['Dim'] == ARGS.dim)]
  print(df.head())
  # Create plot
  sns.set_style('whitegrid')
  sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
  plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
  plt.ylim(0.4, 1.0)
  plt.ylabel("Accuracy")
  plt.xlim(1, 32)
  plt.xlabel("# of steps")
  showsave_plot()

def eval_len(item='pl'):
  """Evaluate model on increasing constant and predicate lengths."""
  # Evaluate model on increasing length test data
  model = create_model(iterations=ARGS.iterations, training=True)
  training, _, run = MODEL_FNAME.split('_')
  for s in ['pl', 'cl']:
    results = list()
    for i in range(2, 65):
      dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                ARGS.batch_size, pad=ARGS.pad, verbose=False)
      results.append(model.evaluate_generator(dgen)[1])
    print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')

def plot_len():
  """Plot increasing length results."""
  # Plot the results
  df = pd.read_csv("len_results.csv")
  df['Mean'] = df.iloc[:,5:].mean(axis=1)
  # Get maximum run based on mean
  idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
  df = df.loc[idx]
  df = df.drop(columns=['Mean'])
  df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
  df['Len'] = df['Len'].astype(int)
  df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
  print(df.head())
  # Create plot
```

```
  sns.set_style('whitegrid')
  sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
  plt.ylim(0.4, 1.0)
  plt.ylabel("Accuracy")
  plt.xlim(2, 64)
  plt.xlabel("Length of symbols")
  showsave_plot()

def plot_dim():
  """Plot increasing dimension over increasing difficulty."""
  df = pd.read_csv("results.csv")
  df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
  print(df.head())
  sns.set_style('whitegrid')
  sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
               hue_order=['validation', 'easy', 'medium', 'hard'])
  plt.ylim(0.5, 1.0)
  plt.ylabel("Mean Accuracy")
  plt.xlim(32, 128)
  plt.xlabel("Dimension")
  plt.legend(loc='upper left')
  showsave_plot()

def plot_training():
  """Plot training method on mean accuracy per test set."""
  df = pd.read_csv("results.csv")
  df = df[(df['Model'] == ARGS.model)]
  print(df.head())
  sns.set_style('whitegrid')
  sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
  plt.ylabel("Mean Accuracy")
  plt.ylim(0.5, 1.0)
  showsave_plot()

def get_pca(context, model, dims=2):
  """Plot the PCA of predicate embeddings."""
  dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
                  train=False, shuffle=False, zeropad=False)
  embds = model.predict_generator(dgen)
  embds = embds.squeeze()
  pca = PCA(dims)
  embds = pca.fit_transform(embds)
  print("TRANSFORMED:", embds)
  print("VAR:", pca.explained_variance_ratio_)
  return embds

def offset(x):
  """Calculate offset for annotation."""
  r = np.random.randint(10, 30)
  return -r if x > 0 else r

def plot_single_preds():
  """Plot embeddings of single word predicates."""
  model = create_model(pca=True)
```

```
  syms = "abcdefghijklmnopqrstuvwxyz"
  ctx, splits = list(), list()
  preds = list("pqrv")
  preds.extend(['' .join([e]*2) for e in preds])
  for p in preds:
    for c in syms:
      ctx.append("{}({}).".format(p,c))
    splits.append(len(ctx))
  embds = get_pca(ctx, model, dims=len(preds))
  from mpl_toolkits.mplot3d import Axes3D
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  prev_sp = 0
  for sp in splits:
    x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
    ax.scatter(x, y, z, depthshade=False)
    for i in map(syms.index, "fdgm"):
      ax.text(x[i], y[i], z[i], ctx[prev_sp+i])
    prev_sp = sp
  showsave_plot()

def plot_single_word():
  """Plot embeddings of single word predicates."""
  model = create_model(pca=True)
  #syms = "abcdefghijklmnopqrstuvwxyz"
  syms = WORD_INDEX.keys()
  ctx, splits = list(), list()
  # preds = list("pqrv")
  # preds.extend(['' .join([e]*2) for e in preds])
  for i in WORD_INDEX.keys():
    ctx.append(i)
    splits.append(len(ctx))
  # for p in preds:
  #   for c in syms:
  #     ctx.append("{}({}).".format(p,c))
  #   splits.append(len(ctx))
  embds = get_pca(ctx, model, dims=64)
  from mpl_toolkits.mplot3d import Axes3D
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  prev_sp = 0
  for sp in splits:
    x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
    ax.scatter(x, y, z, depthshade=False)
    for i in map(syms.index, ["blue","someone","are","bob"]):
      ax.text(x[i], y[i], z[i], ctx[prev_sp+i])
    prev_sp = sp
  showsave_plot()

def plot_pred_saturation():
  """Plot predicate embedding saturation."""
  model = create_model(pca=True)
  ctx, splits = list(), list()
  for i in range(2, 65):
```

```
    p = "".join(["p"]*i)
    ctx.append("{}(a).".format(p))
    splits.append(len(ctx))
  embds = get_pca(ctx, model)
  plt.scatter(embds[::2, 0], embds[::2, 1])
  plt.scatter(embds[1::2, 0], embds[1::2, 1])
  prev_sp = 0
  for i, sp in enumerate(splits):
    pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
    count = pred.count('p')
    if count <= 6:
      xf, yf = offset(x), offset(y)
        plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    elif i % 3 == 0 and i < 50 or i == len(splits)-1 or i == len(splits)-2:
      pred = str(count)+"*p(a)"
      xf, yf = offset(x), offset(y)
        plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    prev_sp = sp
  # Plot contour
  plt.xlim(-2, 2)
  xmin, xmax = plt.xlim()
  X = np.linspace(xmin, xmax, 40)
  ymin, ymax = plt.ylim()
  Y = np.linspace(ymin, ymax, 40)
  X, Y = np.meshgrid(X, Y)
  Z = np.sqrt((X-embds[-1,0])**2 + (Y-embds[-1,1])**2)
  plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
  Z = np.sqrt((X-embds[-2,0])**2 + (Y-embds[-2,1])**2)
  plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
  showsave_plot()

def plot_template(preds, temps):
  """Plot PCA of templates with given predicates."""
  model = create_model(pca=True)
  ctx, splits = list(), list()
  for t in temps:
    for p in preds:
      ctx.append(t.format(p))
    splits.append(len(ctx))
  embds = get_pca(ctx, model)
  prev_sp = 0
  for sp in splits:
    plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
    prev_sp = sp
    pred, x, y = ctx[sp-1], embds[sp-1, 0], embds[sp-1, 1]
    xf, yf = offset(x), offset(y)
    plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle': '-'})
  showsave_plot()

def plot_struct_preds():
  """Plot embeddings of different structural predicates."""
  ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
```

```
  temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
           "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
           "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
           "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
  plot_template(ps, temps)

def plot_rules():
  """Plot embeddings of rules."""
  ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
  temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
           "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
           "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
           "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
           "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
  plot_template(ps, temps)

def plot_attention():
  """Plot attention vector over given context."""
  model = create_model(iterations=ARGS.iterations, training=False)
  ctxs = ["If someone is cold then they are red. if someone is red then they are smart. Fiona is cold."]
  ctxs = [s.lower().replace(",","") for s in ctxs]
  q = "Fiona is smart."
  q = q .lower()
  q = q.replace(".","")
  q_list = [q]
  fig, axes = plt.subplots(1,1)
  for i, ctx in enumerate(ctxs):
    for j, t in enumerate(q_list):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
      out = model.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      #sims = sims.reshape(36,1)
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes.get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1, ARGS.iterations+1),
                  linewidths=0.5, square=True, cbar=False, ax=axes)
      axes.set_xlabel(q+" "+str(out) if j % 2 == 0 else "p(b) "+str(out))
  #plt.tight_layout()
  showsave_plot()

def plot_dual_attention():
  """Plot dual attention vectors of 2 models over given context."""
  modelbw = build_model("imasm", ARGS.model_dir+"curr_imasm64.h5",
                        char_size=len(CHAR_IDX)+1,
                        dim=ARGS.dim, iterations=ARGS.iterations,
                        training=False)
  modelfw = build_model("fwimarsm", ARGS.model_dir+"curr_fwimarsm64.h5",
                        char_size=len(CHAR_IDX)+1,
                        dim=ARGS.dim, iterations=ARGS.iterations,
```

```
                        training=False)
  ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
          "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
          "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
  fig, axes = plt.subplots(1, 6)
  # Plot the attention
  for i, ctx in enumerate(ctxs):
    for j, m in enumerate([modelbw, modelfw]):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
      out = m.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes[i*2+j].get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1,5), linewidths=0.5, square=True, cbar=False, ax=axes[i*2+j])
      # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
      axes[i*2+j].set_xlabel("backward" if j % 2 == 0 else "forward")
  plt.tight_layout()
  showsave_plot()

if __name__ == '__main__':
  globals()[ARGS.function]()

==== eval_conceptrule2.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_conceptrule import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()
```

```
if ARGS.outf:
  import matplotlib
  matplotlib.use("Agg") # Bypass X server
import matplotlib.pyplot as plt
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)

def create_model(**kwargs):
  """Create model from global arguments."""
  # Load in the model
  model = build_model(MODEL_NAME, MODEL_WF,
                      char_size=len(CHAR_IDX)+1,
                      dim=ARGS.dim,
                      **kwargs)
  if ARGS.summary:
    model.summary()
  return model

def evaluate():
  """Evaluate model on each test data."""
  model = create_model(iterations=ARGS.iterations, training=True)
  # training, run, glove, pararule, num = MODEL_FNAME.split('_')
  #training, run, glove, pararule = MODEL_FNAME.split('_')
  name_list = []
  name_list = MODEL_FNAME.split('_')

  for s in ['test_task0.jsonl','test_task1.jsonl','val_task0.jsonl','val_task1.jsonl']:
           #for      s      in      ['test_depth_1.jsonl',      'test_depth_2.jsonl',      'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
  # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
  #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
    results = list()
         dgen  =  LogicSeq.from_file("data/ConceptRules2_full_split/test/"+s,  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
    _, acc = model.evaluate_generator(dgen) # [loss, acc]
    results.append(acc)
    print(name_list[0], ARGS.model, ARGS.dim, s, name_list[1], acc, sep=',')
  #print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')

def showsave_plot():
  """Show or save plot."""
  if ARGS.outf:
    plt.savefig(ARGS.outf, bbox_inches='tight')
  else:
    plt.show()

def eval_nstep():
```

```
  """Evaluate model on nstep deduction."""
  # Evaluate model on every nstep test data
  results = list()
  for i in range(1, 33):
    dgen = LogicSeq.from_file("data/test_nstep{}.txt".format(i), ARGS.batch_size, pad=ARGS.pad, verbose=False)
    model = create_model(iterations=max(ARGS.iterations, i+1), training=True)
    results.append(model.evaluate_generator(dgen)[1])
  training, _, run = MODEL_FNAME.split('_')
  print(training, ARGS.model, ARGS.dim, run, *results, sep=',')

def plot_nstep():
  """Plot nstep results."""
  # Plot the results
  df = pd.read_csv("nstep_results.csv")
  # Get maximum run based on mean
  df['Mean'] = df.iloc[:,4:].mean(axis=1)
  idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
  df = df.loc[idx]
  df = df.drop(columns=['Mean'])
  df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
  df['NStep'] = df['NStep'].astype(int)
  df = df[(df['Dim'] == ARGS.dim)]
  print(df.head())
  # Create plot
  sns.set_style('whitegrid')
  sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
  plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
  plt.ylim(0.4, 1.0)
  plt.ylabel("Accuracy")
  plt.xlim(1, 32)
  plt.xlabel("# of steps")
  showsave_plot()

def eval_len(item='pl'):
  """Evaluate model on increasing constant and predicate lengths."""
  # Evaluate model on increasing length test data
  model = create_model(iterations=ARGS.iterations, training=True)
  training, _, run = MODEL_FNAME.split('_')
  for s in ['pl', 'cl']:
    results = list()
    for i in range(2, 65):
      dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                ARGS.batch_size, pad=ARGS.pad, verbose=False)
      results.append(model.evaluate_generator(dgen)[1])
    print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')

def plot_len():
  """Plot increasing length results."""
  # Plot the results
  df = pd.read_csv("len_results.csv")
  df['Mean'] = df.iloc[:,5:].mean(axis=1)
  # Get maximum run based on mean
  idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
  df = df.loc[idx]
```

```
  df = df.drop(columns=['Mean'])
  df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
  df['Len'] = df['Len'].astype(int)
  df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
  print(df.head())
  # Create plot
  sns.set_style('whitegrid')
  sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
  plt.ylim(0.4, 1.0)
  plt.ylabel("Accuracy")
  plt.xlim(2, 64)
  plt.xlabel("Length of symbols")
  showsave_plot()

def plot_dim():
  """Plot increasing dimension over increasing difficulty."""
  df = pd.read_csv("results.csv")
  df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
  print(df.head())
  sns.set_style('whitegrid')
  sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
               hue_order=['validation', 'easy', 'medium', 'hard'])
  plt.ylim(0.5, 1.0)
  plt.ylabel("Mean Accuracy")
  plt.xlim(32, 128)
  plt.xlabel("Dimension")
  plt.legend(loc='upper left')
  showsave_plot()

def plot_training():
  """Plot training method on mean accuracy per test set."""
  df = pd.read_csv("results.csv")
  df = df[(df['Model'] == ARGS.model)]
  print(df.head())
  sns.set_style('whitegrid')
  sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
  plt.ylabel("Mean Accuracy")
  plt.ylim(0.5, 1.0)
  showsave_plot()

def get_pca(context, model, dims=2):
  """Plot the PCA of predicate embeddings."""
  dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
                  train=False, shuffle=False, zeropad=False)
  embds = model.predict_generator(dgen)
  embds = embds.squeeze()
  pca = PCA(dims)
  embds = pca.fit_transform(embds)
  print("TRANSFORMED:", embds)
  print("VAR:", pca.explained_variance_ratio_)
  return embds

def offset(x):
  """Calculate offset for annotation."""
```

```
  r = np.random.randint(10, 30)
  return -r if x > 0 else r

def plot_single_preds():
  """Plot embeddings of single word predicates."""
  model = create_model(pca=True)
  syms = "abcdefghijklmnopqrstuvwxyz"
  ctx, splits = list(), list()
  preds = list("pqrv")
  preds.extend(['',join([e]*2) for e in preds])
  for p in preds:
    for c in syms:
      ctx.append("{}({}).".format(p,c))
    splits.append(len(ctx))
  embds = get_pca(ctx, model, dims=len(preds))
  from mpl_toolkits.mplot3d import Axes3D
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  prev_sp = 0
  for sp in splits:
    x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
    ax.scatter(x, y, z, depthshade=False)
    for i in map(syms.index, "fdgm"):
      ax.text(x[i], y[i], z[i], ctx[prev_sp+i])
    prev_sp = sp
  showsave_plot()

def plot_single_word():
  """Plot embeddings of single word predicates."""
  model = create_model(pca=True)
  #syms = "abcdefghijklmnopqrstuvwxyz"
  syms = WORD_INDEX.keys()
  ctx, splits = list(), list()
  # preds = list("pqrv")
  # preds.extend(['',join([e]*2) for e in preds])
  for i in WORD_INDEX.keys():
    ctx.append(i)
    splits.append(len(ctx))
  # for p in preds:
  #   for c in syms:
  #     ctx.append("{}({}).".format(p,c))
  #   splits.append(len(ctx))
  embds = get_pca(ctx, model, dims=64)
  from mpl_toolkits.mplot3d import Axes3D
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  prev_sp = 0
  for sp in splits:
    x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
    ax.scatter(x, y, z, depthshade=False)
    for i in map(syms.index, ["blue","someone","are","bob"]):
      ax.text(x[i], y[i], z[i], ctx[prev_sp+i])
    prev_sp = sp
  showsave_plot()
```

```
def plot_pred_saturation():
  """Plot predicate embedding saturation."""
  model = create_model(pca=True)
  ctx, splits = list(), list()
  for i in range(2, 65):
    p = "".join(["p"]*i)
    ctx.append("{}(a).".format(p))
    splits.append(len(ctx))
  embds = get_pca(ctx, model)
  plt.scatter(embds[::2, 0], embds[::2, 1])
  plt.scatter(embds[1::2, 0], embds[1::2, 1])
  prev_sp = 0
  for i, sp in enumerate(splits):
    pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
    count = pred.count('p')
    if count <= 6:
      xf, yf = offset(x), offset(y)
        plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    elif i % 3 == 0 and i < 50 or i == len(splits)-1 or i == len(splits)-2:
      pred = str(count)+"*p(a)"
      xf, yf = offset(x), offset(y)
        plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    prev_sp = sp
  # Plot contour
  plt.xlim(-2, 2)
  xmin, xmax = plt.xlim()
  X = np.linspace(xmin, xmax, 40)
  ymin, ymax = plt.ylim()
  Y = np.linspace(ymin, ymax, 40)
  X, Y = np.meshgrid(X, Y)
  Z = np.sqrt((X-embds[-1,0])**2 + (Y-embds[-1,1])**2)
  plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
  Z = np.sqrt((X-embds[-2,0])**2 + (Y-embds[-2,1])**2)
  plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
  showsave_plot()

def plot_template(preds, temps):
  """Plot PCA of templates with given predicates."""
  model = create_model(pca=True)
  ctx, splits = list(), list()
  for t in temps:
    for p in preds:
      ctx.append(t.format(p))
    splits.append(len(ctx))
  embds = get_pca(ctx, model)
  prev_sp = 0
  for sp in splits:
    plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
    prev_sp = sp
    pred, x, y = ctx[sp-1], embds[sp-1, 0], embds[sp-1, 1]
    xf, yf = offset(x), offset(y)
```

```
    plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle': '-'})
  showsave_plot()

def plot_struct_preds():
  """Plot embeddings of different structural predicates."""
  ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
  temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
           "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
           "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
           "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
  plot_template(ps, temps)

def plot_rules():
  """Plot embeddings of rules."""
  ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
  temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
           "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
           "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
           "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
           "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
  plot_template(ps, temps)

def plot_attention():
  """Plot attention vector over given context."""
  model = create_model(iterations=ARGS.iterations, training=False)
  ctxs = ["City is a part of province. Curl iron is situated at entrance to school. Table is not situated at home with piano. City is situated at germany. Chair is not situated at home with piano. House is not a home. Table is situated at house. Chair is situated at table. Poop is situated at entrance to school. House is situated at city. Chair is not situated at major city. Curl iron is not situated at germany."]
  ctxs = [s.lower().replace(",","") for s in ctxs]
  q = "Chair is situated at house."
  q = q .lower()
  q = q.replace(".","")
  q_list = [q]
  fig, axes = plt.subplots(1,1)
  for i, ctx in enumerate(ctxs):
    for j, t in enumerate(q_list):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
      out = model.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      #sims = sims.reshape(36,1)
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes.get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1, ARGS.iterations+1),
                  linewidths=0.5, square=True, cbar=False, ax=axes)
      axes.set_xlabel(q+" "+str(out) if j % 2 == 0 else "p(b) "+str(out))
  #plt.tight_layout()
  #showsave_plot()
  plt.savefig('conceptruleV2_full_depth0.png', bbox_inches='tight')
```

```
def plot_dual_attention():
  """Plot dual attention vectors of 2 models over given context."""
  modelbw = build_model("imasm", ARGS.model_dir+"curr_imasm64.h5",
                        char_size=len(CHAR_IDX)+1,
                        dim=ARGS.dim, iterations=ARGS.iterations,
                        training=False)
  modelfw = build_model("fwimarsm", ARGS.model_dir+"curr_fwimarsm64.h5",
                        char_size=len(CHAR_IDX)+1,
                        dim=ARGS.dim, iterations=ARGS.iterations,
                        training=False)
  ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
          "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
          "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
  fig, axes = plt.subplots(1, 6)
  # Plot the attention
  for i, ctx in enumerate(ctxs):
    for j, m in enumerate([modelbw, modelfw]):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
      out = m.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes[i*2+j].get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1,5), linewidths=0.5, square=True, cbar=False, ax=axes[i*2+j])
      # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
      axes[i*2+j].set_xlabel("backward" if j % 2 == 0 else "forward")
  plt.tight_layout()
  showsave_plot()

if __name__ == '__main__':
  globals()[ARGS.function]()

==== eval_conceptrule2_csv.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_conceptrule_csv import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
```

```
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()

if ARGS.outf:
    import matplotlib

    matplotlib.use("Agg")  # Bypass X server
import matplotlib.pyplot as plt
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)



def create_model(**kwargs):
    """Create model from global arguments."""
    # Load in the model
    model = build_model(MODEL_NAME, MODEL_WF,
                        char_size=len(CHAR_IDX) + 1,
                        dim=ARGS.dim,
                        **kwargs)
    if ARGS.summary:
        model.summary()
    return model



def evaluate():
    """Evaluate model on each test data."""
    model = create_model(iterations=ARGS.iterations, training=True)
    # training, run, glove, pararule, num = MODEL_FNAME.split('_')
    name_list = []
    # training, run, glove, pararule, batch = MODEL_FNAME.split('_')
    name_list = MODEL_FNAME.split('_')

    for s in ['test_task0.csv', 'test_task1.csv', 'test_task2.csv', 'test_task3.csv', 'test_task4.csv',
              'test_task5.csv', 'test_task6.csv']:
                        #   for   s   in   ['test_depth_1.jsonl',   'test_depth_2.jsonl',   'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
        # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
        #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
        results = list()
          dgen = LogicSeq.from_file("data/ConceptRules2_full_split/test/" + s, ARGS.batch_size, pad=ARGS.pad,
```

```
verbose=False)
        _, acc = model.evaluate_generator(dgen)  # [loss, acc]
        results.append(acc)
        print(name_list[0], ARGS.model, ARGS.dim, s, name_list[1], acc, sep=',')
        #print(training, ARGS.model, ARGS.dim, s, run, acc, sep=',')
    # print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def showsave_plot():
    """Show or save plot."""
    if ARGS.outf:
        plt.savefig(ARGS.outf, bbox_inches='tight')
    else:
        plt.show()


def eval_nstep():
    """Evaluate model on nstep deduction."""
    # Evaluate model on every nstep test data
    results = list()
    for i in range(1, 33):
                dgen  =  LogicSeq.from_file("data/test_nstep{}.txt".format(i),  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
        model = create_model(iterations=max(ARGS.iterations, i + 1), training=True)
        results.append(model.evaluate_generator(dgen)[1])
    training, _, run = MODEL_FNAME.split('_')
    print(training, ARGS.model, ARGS.dim, run, *results, sep=',')


def plot_nstep():
    """Plot nstep results."""
    # Plot the results
    df = pd.read_csv("nstep_results.csv")
    # Get maximum run based on mean
    df['Mean'] = df.iloc[:, 4:].mean(axis=1)
    idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
    df['NStep'] = df['NStep'].astype(int)
    df = df[(df['Dim'] == ARGS.dim)]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
    plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(1, 32)
    plt.xlabel("# of steps")
    showsave_plot()


def eval_len(item='pl'):
```

```
    """Evaluate model on increasing constant and predicate lengths."""
    # Evaluate model on increasing length test data
    model = create_model(iterations=ARGS.iterations, training=True)
    training, _, run = MODEL_FNAME.split('_')
    for s in ['pl', 'cl']:
        results = list()
        for i in range(2, 65):
            dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                      ARGS.batch_size, pad=ARGS.pad, verbose=False)
            results.append(model.evaluate_generator(dgen)[1])
        print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def plot_len():
    """Plot increasing length results."""
    # Plot the results
    df = pd.read_csv("len_results.csv")
    df['Mean'] = df.iloc[:, 5:].mean(axis=1)
    # Get maximum run based on mean
    idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
    df['Len'] = df['Len'].astype(int)
    df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(2, 64)
    plt.xlabel("Length of symbols")
    showsave_plot()


def plot_dim():
    """Plot increasing dimension over increasing difficulty."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
    print(df.head())
    sns.set_style('whitegrid')
    sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
                 hue_order=['validation', 'easy', 'medium', 'hard'])
    plt.ylim(0.5, 1.0)
    plt.ylabel("Mean Accuracy")
    plt.xlim(32, 128)
    plt.xlabel("Dimension")
    plt.legend(loc='upper left')
    showsave_plot()


def plot_training():
    """Plot training method on mean accuracy per test set."""
```

```
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model)]
    print(df.head())
    sns.set_style('whitegrid')
    sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
    plt.ylabel("Mean Accuracy")
    plt.ylim(0.5, 1.0)
    showsave_plot()


def get_pca(context, model, dims=2):
    """Plot the PCA of predicate embeddings."""
    dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
                    train=False, shuffle=False, zeropad=False)
    embds = model.predict_generator(dgen)
    embds = embds.squeeze()
    pca = PCA(dims)
    embds = pca.fit_transform(embds)
    print("TRANSFORMED:", embds)
    print("VAR:", pca.explained_variance_ratio_)
    return embds


def offset(x):
    """Calculate offset for annotation."""
    r = np.random.randint(10, 30)
    return -r if x > 0 else r


def plot_single_preds():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    syms = "abcdefghijklmnopqrstuvwxyz"
    ctx, splits = list(), list()
    preds = list("pqrv")
    preds.extend(['' .join([e] * 2) for e in preds])
    for p in preds:
        for c in syms:
            ctx.append("{}({}).".format(p, c))
        splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=len(preds))
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, "fdgm"):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()
```

```
def plot_single_word():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    # syms = "abcdefghijklmnopqrstuvwxyz"
    syms = WORD_INDEX.keys()
    ctx, splits = list(), list()
    # preds = list("pqrv")
    # preds.extend(['',join([e]*2) for e in preds])
    for i in WORD_INDEX.keys():
        ctx.append(i)
        splits.append(len(ctx))
    # for p in preds:
    #   for c in syms:
    #     ctx.append("{}({}).".format(p,c))
    #   splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=64)
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, ["blue", "someone", "are", "bob"]):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_pred_saturation():
    """Plot predicate embedding saturation."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for i in range(2, 65):
        p = "".join(["p"] * i)
        ctx.append("{}(a).".format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    plt.scatter(embds[::2, 0], embds[::2, 1])
    plt.scatter(embds[1::2, 0], embds[1::2, 1])
    prev_sp = 0
    for i, sp in enumerate(splits):
        pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
        count = pred.count('p')
        if count <= 6:
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        elif i % 3 == 0 and i < 50 or i == len(splits) - 1 or i == len(splits) - 2:
            pred = str(count) + "*p(a)"
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        prev_sp = sp
```

```
    # Plot contour
    plt.xlim(-2, 2)
    xmin, xmax = plt.xlim()
    X = np.linspace(xmin, xmax, 40)
    ymin, ymax = plt.ylim()
    Y = np.linspace(ymin, ymax, 40)
    X, Y = np.meshgrid(X, Y)
    Z = np.sqrt((X - embds[-1, 0]) ** 2 + (Y - embds[-1, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    Z = np.sqrt((X - embds[-2, 0]) ** 2 + (Y - embds[-2, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    showsave_plot()


def plot_template(preds, temps):
    """Plot PCA of templates with given predicates."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for t in temps:
        for p in preds:
            ctx.append(t.format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    prev_sp = 0
    for sp in splits:
        plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
        prev_sp = sp
        pred, x, y = ctx[sp - 1], embds[sp - 1, 0], embds[sp - 1, 1]
        xf, yf = offset(x), offset(y)
          plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    showsave_plot()


def plot_struct_preds():
    """Plot embeddings of different structural predicates."""
    ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
             "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
    plot_template(ps, temps)


def plot_rules():
    """Plot embeddings of rules."""
    ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
             "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
             "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
    plot_template(ps, temps)
```

```
def plot_attention():
    """Plot attention vector over given context."""
    model = create_model(iterations=ARGS.iterations, training=False)
    ctxs = ["If someone is cold then they are red. if someone is red then they are smart. Fiona is cold."]
    ctxs = [s.lower().replace(",", "") for s in ctxs]
    q = "Fiona is smart."
    q = q.lower()
    q = q.replace(".", "")
    q_list = [q]
    fig, axes = plt.subplots(1, 1)
    for i, ctx in enumerate(ctxs):
        for j, t in enumerate(q_list):
            rs = ctx.split('.')[:-1]
            dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
            out = model.predict_generator(dgen)
            sims = out[:-1]
            out = np.round(np.asscalar(out[-1]), 2)
            sims = np.stack(sims, axis=0).squeeze()
            sims = sims.T
            # sims = sims.reshape(36,1)
            ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
            axes.get_xaxis().set_ticks_position('top')
            sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                        xticklabels=range(1, ARGS.iterations + 1),
                        linewidths=0.5, square=True, cbar=False, ax=axes)
            axes.set_xlabel(q + " " + str(out) if j % 2 == 0 else "p(b) " + str(out))
    # plt.tight_layout()
    showsave_plot()


def plot_dual_attention():
    """Plot dual attention vectors of 2 models over given context."""
    modelbw = build_model("imasm", ARGS.model_dir + "curr_imasm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    modelfw = build_model("fwimarsm", ARGS.model_dir + "curr_fwimarsm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
            "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
            "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
    fig, axes = plt.subplots(1, 6)
    # Plot the attention
    for i, ctx in enumerate(ctxs):
        for j, m in enumerate([modelbw, modelfw]):
            rs = ctx.split('.')[:-1]
            dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
            out = m.predict_generator(dgen)
            sims = out[:-1]
            out = np.round(np.asscalar(out[-1]), 2)
            sims = np.stack(sims, axis=0).squeeze()
```

```
            sims = sims.T
            ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
            axes[i * 2 + j].get_xaxis().set_ticks_position('top')
            sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                        xticklabels=range(1, 5), linewidths=0.5, square=True, cbar=False, ax=axes[i * 2 + j])
            # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
            axes[i * 2 + j].set_xlabel("backward" if j % 2 == 0 else "forward")
    plt.tight_layout()
    showsave_plot()


if __name__ == '__main__':
    globals()[ARGS.function]()
```

```
==== eval_conceptrule2_full_filter_csv.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_conceptrule_csv import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()

if ARGS.outf:
    import matplotlib

    matplotlib.use("Agg")  # Bypass X server
import matplotlib.pyplot as plt
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
```

```
np.set_printoptions(suppress=True)


def create_model(**kwargs):
    """Create model from global arguments."""
    # Load in the model
    model = build_model(MODEL_NAME, MODEL_WF,
                        char_size=len(CHAR_IDX) + 1,
                        dim=ARGS.dim,
                        **kwargs)
    if ARGS.summary:
        model.summary()
    return model


def evaluate():
    """Evaluate model on each test data."""
    model = create_model(iterations=ARGS.iterations, training=True)
    # training, run, glove, pararule, num = MODEL_FNAME.split('_')
    name_list = []
    # training, run, glove, pararule, batch = MODEL_FNAME.split('_')
    name_list = MODEL_FNAME.split('_')

    for s in ['test_task_CWA0.csv', 'test_task_CWA1.csv', 'test_task_no_CWA0.csv', 'test_task_no_CWA1.csv']:
                        #   for   s   in   ['test_depth_1.jsonl',   'test_depth_2.jsonl',   'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
        # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
        #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
        results = list()
                dgen  =  LogicSeq.from_file("data/ConceptRulesV2/test/"  +  s,  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
        _, acc = model.evaluate_generator(dgen)  # [loss, acc]
        results.append(acc)
        print(name_list[0], ARGS.model, ARGS.dim, s, name_list[1], acc, sep=',')
        #print(training, ARGS.model, ARGS.dim, s, run, acc, sep=',')
    # print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def showsave_plot():
    """Show or save plot."""
    if ARGS.outf:
        plt.savefig(ARGS.outf, bbox_inches='tight')
    else:
        plt.show()


def eval_nstep():
    """Evaluate model on nstep deduction."""
    # Evaluate model on every nstep test data
    results = list()
    for i in range(1, 33):
                dgen  =  LogicSeq.from_file("data/test_nstep{}.txt".format(i),  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
        model = create_model(iterations=max(ARGS.iterations, i + 1), training=True)
```

```
        results.append(model.evaluate_generator(dgen)[1])
    training, _, run = MODEL_FNAME.split('_')
    print(training, ARGS.model, ARGS.dim, run, *results, sep=',')


def plot_nstep():
    """Plot nstep results."""
    # Plot the results
    df = pd.read_csv("nstep_results.csv")
    # Get maximum run based on mean
    df['Mean'] = df.iloc[:, 4:].mean(axis=1)
    idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
    df['NStep'] = df['NStep'].astype(int)
    df = df[(df['Dim'] == ARGS.dim)]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
    plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(1, 32)
    plt.xlabel("# of steps")
    showsave_plot()


def eval_len(item='pl'):
    """Evaluate model on increasing constant and predicate lengths."""
    # Evaluate model on increasing length test data
    model = create_model(iterations=ARGS.iterations, training=True)
    training, _, run = MODEL_FNAME.split('_')
    for s in ['pl', 'cl']:
        results = list()
        for i in range(2, 65):
            dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                      ARGS.batch_size, pad=ARGS.pad, verbose=False)
            results.append(model.evaluate_generator(dgen)[1])
        print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def plot_len():
    """Plot increasing length results."""
    # Plot the results
    df = pd.read_csv("len_results.csv")
    df['Mean'] = df.iloc[:, 5:].mean(axis=1)
    # Get maximum run based on mean
    idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
    df['Len'] = df['Len'].astype(int)
```

```
    df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(2, 64)
    plt.xlabel("Length of symbols")
    showsave_plot()


def plot_dim():
    """Plot increasing dimension over increasing difficulty."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
    print(df.head())
    sns.set_style('whitegrid')
    sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
                 hue_order=['validation', 'easy', 'medium', 'hard'])
    plt.ylim(0.5, 1.0)
    plt.ylabel("Mean Accuracy")
    plt.xlim(32, 128)
    plt.xlabel("Dimension")
    plt.legend(loc='upper left')
    showsave_plot()


def plot_training():
    """Plot training method on mean accuracy per test set."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model)]
    print(df.head())
    sns.set_style('whitegrid')
    sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
    plt.ylabel("Mean Accuracy")
    plt.ylim(0.5, 1.0)
    showsave_plot()


def get_pca(context, model, dims=2):
    """Plot the PCA of predicate embeddings."""
    dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
                    train=False, shuffle=False, zeropad=False)
    embds = model.predict_generator(dgen)
    embds = embds.squeeze()
    pca = PCA(dims)
    embds = pca.fit_transform(embds)
    print("TRANSFORMED:", embds)
    print("VAR:", pca.explained_variance_ratio_)
    return embds


def offset(x):
```

```
    """Calculate offset for annotation."""
    r = np.random.randint(10, 30)
    return -r if x > 0 else r


def plot_single_preds():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    syms = "abcdefghijklmnopqrstuvwxyz"
    ctx, splits = list(), list()
    preds = list("pqrv")
    preds.extend(['' .join([e] * 2) for e in preds])
    for p in preds:
        for c in syms:
            ctx.append("{}({}).".format(p, c))
        splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=len(preds))
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, "fdgm"):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_single_word():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    # syms = "abcdefghijklmnopqrstuvwxyz"
    syms = WORD_INDEX.keys()
    ctx, splits = list(), list()
    # preds = list("pqrv")
    # preds.extend(['' .join([e]*2) for e in preds])
    for i in WORD_INDEX.keys():
        ctx.append(i)
        splits.append(len(ctx))
    # for p in preds:
    #   for c in syms:
    #     ctx.append("{}({}).".format(p,c))
    #   splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=64)
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, ["blue", "someone", "are", "bob"]):
```

```
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_pred_saturation():
    """Plot predicate embedding saturation."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for i in range(2, 65):
        p = "".join(["p"] * i)
        ctx.append("{}(a).".format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    plt.scatter(embds[::2, 0], embds[::2, 1])
    plt.scatter(embds[1::2, 0], embds[1::2, 1])
    prev_sp = 0
    for i, sp in enumerate(splits):
        pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
        count = pred.count('p')
        if count <= 6:
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        elif i % 3 == 0 and i < 50 or i == len(splits) - 1 or i == len(splits) - 2:
            pred = str(count) + "*p(a)"
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        prev_sp = sp
    # Plot contour
    plt.xlim(-2, 2)
    xmin, xmax = plt.xlim()
    X = np.linspace(xmin, xmax, 40)
    ymin, ymax = plt.ylim()
    Y = np.linspace(ymin, ymax, 40)
    X, Y = np.meshgrid(X, Y)
    Z = np.sqrt((X - embds[-1, 0]) ** 2 + (Y - embds[-1, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    Z = np.sqrt((X - embds[-2, 0]) ** 2 + (Y - embds[-2, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    showsave_plot()


def plot_template(preds, temps):
    """Plot PCA of templates with given predicates."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for t in temps:
        for p in preds:
            ctx.append(t.format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    prev_sp = 0
```

```
    for sp in splits:
        plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
        prev_sp = sp
        pred, x, y = ctx[sp - 1], embds[sp - 1, 0], embds[sp - 1, 1]
        xf, yf = offset(x), offset(y)
          plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    showsave_plot()


def plot_struct_preds():
    """Plot embeddings of different structural predicates."""
    ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
             "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
    plot_template(ps, temps)


def plot_rules():
    """Plot embeddings of rules."""
    ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
             "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
             "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
    plot_template(ps, temps)


def plot_attention():
  """Plot attention vector over given context."""
  model = create_model(iterations=ARGS.iterations, training=False)
   ctxs = ["City is a part of province. Curl iron is situated at entrance to school. Table is not situated at
home with piano. City is situated at germany. Chair is not situated at home with piano. House is not a home.
Table is situated at house. Chair is situated at table. Poop is situated at entrance to school. House is
situated at city. Chair is not situated at major city. Curl iron is not situated at germany."]
  ctxs = [s.lower().replace(",","") for s in ctxs]
  q = "City is not situated at science."
  q = q .lower()
  q = q.replace(".","")
  q_list = [q]
  fig, axes = plt.subplots(1,1)
  for i, ctx in enumerate(ctxs):
    for j, t in enumerate(q_list):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
      out = model.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      #sims = sims.reshape(36,1)
```

```
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes.get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1, ARGS.iterations+1),
                  linewidths=0.5, square=True, cbar=False, ax=axes)
      axes.set_xlabel(q+" "+str(out) if j % 2 == 0 else "p(b) "+str(out))
  #plt.tight_layout()
  #showsave_plot()
  plt.savefig('Depth-0_full_CWA_case_r.png', bbox_inches='tight')


def plot_dual_attention():
    """Plot dual attention vectors of 2 models over given context."""
    modelbw = build_model("imasm", ARGS.model_dir + "curr_imasm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    modelfw = build_model("fwimarsm", ARGS.model_dir + "curr_fwimarsm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
            "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
            "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
    fig, axes = plt.subplots(1, 6)
    # Plot the attention
    for i, ctx in enumerate(ctxs):
        for j, m in enumerate([modelbw, modelfw]):
            rs = ctx.split('.')[:-1]
            dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
            out = m.predict_generator(dgen)
            sims = out[:-1]
            out = np.round(np.asscalar(out[-1]), 2)
            sims = np.stack(sims, axis=0).squeeze()
            sims = sims.T
            ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
            axes[i * 2 + j].get_xaxis().set_ticks_position('top')
            sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                        xticklabels=range(1, 5), linewidths=0.5, square=True, cbar=False, ax=axes[i * 2 + j])
            # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
            axes[i * 2 + j].set_xlabel("backward" if j % 2 == 0 else "forward")
    plt.tight_layout()
    showsave_plot()


if __name__ == '__main__':
    globals()[ARGS.function]()

==== eval_conceptrule2_simplified_csv.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
```

```
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_conceptrule_csv import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()

if ARGS.outf:
    import matplotlib

    matplotlib.use("Agg")  # Bypass X server
import matplotlib.pyplot as plt
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)


def create_model(**kwargs):
    """Create model from global arguments."""
    # Load in the model
    model = build_model(MODEL_NAME, MODEL_WF,
                        char_size=len(CHAR_IDX) + 1,
                        dim=ARGS.dim,
                        **kwargs)
    if ARGS.summary:
        model.summary()
    return model


def evaluate():
    """Evaluate model on each test data."""
    model = create_model(iterations=ARGS.iterations, training=True)
    # training, run, glove, pararule, num = MODEL_FNAME.split('_')
    name_list = []
```

```
    # training, run, glove, pararule, batch = MODEL_FNAME.split('_')
    name_list = MODEL_FNAME.split('_')

    for s in ['test_task0.csv', 'test_task1.csv', 'test_task2.csv', 'test_task3.csv', 'test_task4.csv',
              'test_task5.csv', 'test_task6.csv']:
                        #   for   s   in   ['test_depth_1.jsonl',   'test_depth_2.jsonl',   'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
        # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
        #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
        results = list()
               dgen  =  LogicSeq.from_file("data/ConceptRules2_simplified_split/test/"  +  s,  ARGS.batch_size,
pad=ARGS.pad, verbose=False)
        _, acc = model.evaluate_generator(dgen)  # [loss, acc]
        results.append(acc)
        print(name_list[0], ARGS.model, ARGS.dim, s, name_list[1], acc, sep=',')
        #print(training, ARGS.model, ARGS.dim, s, run, acc, sep=',')
    # print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def showsave_plot():
    """Show or save plot."""
    if ARGS.outf:
        plt.savefig(ARGS.outf, bbox_inches='tight')
    else:
        plt.show()


def eval_nstep():
    """Evaluate model on nstep deduction."""
    # Evaluate model on every nstep test data
    results = list()
    for i in range(1, 33):
                dgen  =  LogicSeq.from_file("data/test_nstep{}.txt".format(i),  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
        model = create_model(iterations=max(ARGS.iterations, i + 1), training=True)
        results.append(model.evaluate_generator(dgen)[1])
    training, _, run = MODEL_FNAME.split('_')
    print(training, ARGS.model, ARGS.dim, run, *results, sep=',')


def plot_nstep():
    """Plot nstep results."""
    # Plot the results
    df = pd.read_csv("nstep_results.csv")
    # Get maximum run based on mean
    df['Mean'] = df.iloc[:, 4:].mean(axis=1)
    idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
    df['NStep'] = df['NStep'].astype(int)
    df = df[(df['Dim'] == ARGS.dim)]
    print(df.head())
    # Create plot
```

```
    sns.set_style('whitegrid')
    sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
    plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(1, 32)
    plt.xlabel("# of steps")
    showsave_plot()


def eval_len(item='pl'):
    """Evaluate model on increasing constant and predicate lengths."""
    # Evaluate model on increasing length test data
    model = create_model(iterations=ARGS.iterations, training=True)
    training, _, run = MODEL_FNAME.split('_')
    for s in ['pl', 'cl']:
        results = list()
        for i in range(2, 65):
            dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                      ARGS.batch_size, pad=ARGS.pad, verbose=False)
            results.append(model.evaluate_generator(dgen)[1])
        print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def plot_len():
    """Plot increasing length results."""
    # Plot the results
    df = pd.read_csv("len_results.csv")
    df['Mean'] = df.iloc[:, 5:].mean(axis=1)
    # Get maximum run based on mean
    idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
    df['Len'] = df['Len'].astype(int)
    df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(2, 64)
    plt.xlabel("Length of symbols")
    showsave_plot()


def plot_dim():
    """Plot increasing dimension over increasing difficulty."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
    print(df.head())
    sns.set_style('whitegrid')
    sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
```

```
                hue_order=['validation', 'easy', 'medium', 'hard'])
    plt.ylim(0.5, 1.0)
    plt.ylabel("Mean Accuracy")
    plt.xlim(32, 128)
    plt.xlabel("Dimension")
    plt.legend(loc='upper left')
    showsave_plot()


def plot_training():
    """Plot training method on mean accuracy per test set."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model)]
    print(df.head())
    sns.set_style('whitegrid')
    sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
    plt.ylabel("Mean Accuracy")
    plt.ylim(0.5, 1.0)
    showsave_plot()


def get_pca(context, model, dims=2):
    """Plot the PCA of predicate embeddings."""
    dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
                    train=False, shuffle=False, zeropad=False)
    embds = model.predict_generator(dgen)
    embds = embds.squeeze()
    pca = PCA(dims)
    embds = pca.fit_transform(embds)
    print("TRANSFORMED:", embds)
    print("VAR:", pca.explained_variance_ratio_)
    return embds


def offset(x):
    """Calculate offset for annotation."""
    r = np.random.randint(10, 30)
    return -r if x > 0 else r


def plot_single_preds():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    syms = "abcdefghijklmnopqrstuvwxyz"
    ctx, splits = list(), list()
    preds = list("pqrv")
    preds.extend([''.join([e] * 2) for e in preds])
    for p in preds:
        for c in syms:
            ctx.append("{}({}).".format(p, c))
        splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=len(preds))
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, "fdgm"):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_single_word():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    # syms = "abcdefghijklmnopqrstuvwxyz"
    syms = WORD_INDEX.keys()
    ctx, splits = list(), list()
    # preds = list("pqrv")
    # preds.extend(['',join([e]*2) for e in preds])
    for i in WORD_INDEX.keys():
        ctx.append(i)
        splits.append(len(ctx))
    # for p in preds:
    #   for c in syms:
    #     ctx.append("{}({}).".format(p,c))
    #   splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=64)
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, ["blue", "someone", "are", "bob"]):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_pred_saturation():
    """Plot predicate embedding saturation."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for i in range(2, 65):
        p = "".join(["p"] * i)
        ctx.append("{}(a).".format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    plt.scatter(embds[::2, 0], embds[::2, 1])
    plt.scatter(embds[1::2, 0], embds[1::2, 1])
    prev_sp = 0
    for i, sp in enumerate(splits):
        pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
```

```
        count = pred.count('p')
        if count <= 6:
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        elif i % 3 == 0 and i < 50 or i == len(splits) - 1 or i == len(splits) - 2:
            pred = str(count) + "*p(a)"
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        prev_sp = sp
    # Plot contour
    plt.xlim(-2, 2)
    xmin, xmax = plt.xlim()
    X = np.linspace(xmin, xmax, 40)
    ymin, ymax = plt.ylim()
    Y = np.linspace(ymin, ymax, 40)
    X, Y = np.meshgrid(X, Y)
    Z = np.sqrt((X - embds[-1, 0]) ** 2 + (Y - embds[-1, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    Z = np.sqrt((X - embds[-2, 0]) ** 2 + (Y - embds[-2, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    showsave_plot()


def plot_template(preds, temps):
    """Plot PCA of templates with given predicates."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for t in temps:
        for p in preds:
            ctx.append(t.format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    prev_sp = 0
    for sp in splits:
        plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
        prev_sp = sp
        pred, x, y = ctx[sp - 1], embds[sp - 1, 0], embds[sp - 1, 1]
        xf, yf = offset(x), offset(y)
          plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    showsave_plot()


def plot_struct_preds():
    """Plot embeddings of different structural predicates."""
    ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
             "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
    plot_template(ps, temps)
```

```
def plot_rules():
    """Plot embeddings of rules."""
    ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
             "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
             "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
    plot_template(ps, temps)


def plot_attention():
    """Plot attention vector over given context."""
    model = create_model(iterations=ARGS.iterations, training=False)
    ctxs = ["If someone is cold then they are red. if someone is red then they are smart. Fiona is cold."]
    ctxs = [s.lower().replace(",", "") for s in ctxs]
    q = "Fiona is smart."
    q = q.lower()
    q = q.replace(".", "")
    q_list = [q]
    fig, axes = plt.subplots(1, 1)
    for i, ctx in enumerate(ctxs):
        for j, t in enumerate(q_list):
            rs = ctx.split('.')[:-1]
            dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
            out = model.predict_generator(dgen)
            sims = out[:-1]
            out = np.round(np.asscalar(out[-1]), 2)
            sims = np.stack(sims, axis=0).squeeze()
            sims = sims.T
            # sims = sims.reshape(36,1)
            ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
            axes.get_xaxis().set_ticks_position('top')
            sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                        xticklabels=range(1, ARGS.iterations + 1),
                        linewidths=0.5, square=True, cbar=False, ax=axes)
            axes.set_xlabel(q + " " + str(out) if j % 2 == 0 else "p(b) " + str(out))
    # plt.tight_layout()
    showsave_plot()


def plot_dual_attention():
    """Plot dual attention vectors of 2 models over given context."""
    modelbw = build_model("imasm", ARGS.model_dir + "curr_imasm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    modelfw = build_model("fwimarsm", ARGS.model_dir + "curr_fwimarsm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
            "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
```

```
            "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
    fig, axes = plt.subplots(1, 6)
    # Plot the attention
    for i, ctx in enumerate(ctxs):
        for j, m in enumerate([modelbw, modelfw]):
            rs = ctx.split('.')[:-1]
            dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
            out = m.predict_generator(dgen)
            sims = out[:-1]
            out = np.round(np.asscalar(out[-1]), 2)
            sims = np.stack(sims, axis=0).squeeze()
            sims = sims.T
            ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
            axes[i * 2 + j].get_xaxis().set_ticks_position('top')
            sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                        xticklabels=range(1, 5), linewidths=0.5, square=True, cbar=False, ax=axes[i * 2 + j])
            # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
            axes[i * 2 + j].set_xlabel("backward" if j % 2 == 0 else "forward")
    plt.tight_layout()
    showsave_plot()


if __name__ == '__main__':
    globals()[ARGS.function]()

==== eval_conceptrule2_simplified_filter_csv.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_conceptrule_csv import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()

if ARGS.outf:
```

```
    import matplotlib

    matplotlib.use("Agg")  # Bypass X server
import matplotlib.pyplot as plt
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)


def create_model(**kwargs):
    """Create model from global arguments."""
    # Load in the model
    model = build_model(MODEL_NAME, MODEL_WF,
                        char_size=len(CHAR_IDX) + 1,
                        dim=ARGS.dim,
                        **kwargs)
    if ARGS.summary:
        model.summary()
    return model


def evaluate():
    """Evaluate model on each test data."""
    model = create_model(iterations=ARGS.iterations, training=True)
    # training, run, glove, pararule, num = MODEL_FNAME.split('_')
    name_list = []
    # training, run, glove, pararule, batch = MODEL_FNAME.split('_')
    name_list = MODEL_FNAME.split('_')

    for s in ['test_task_CWA0.csv', 'test_task_CWA1.csv', 'test_task_no_CWA0.csv', 'test_task_no_CWA1.csv']:
                        #   for   s   in   ['test_depth_1.jsonl',   'test_depth_2.jsonl',   'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
        # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
        #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
        results = list()
                dgen  =  LogicSeq.from_file("data/ConceptRulesV2/test/"  +  s,  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
        _, acc = model.evaluate_generator(dgen)  # [loss, acc]
        results.append(acc)
        print(name_list[0], ARGS.model, ARGS.dim, s, name_list[1], acc, sep=',')
        #print(training, ARGS.model, ARGS.dim, s, run, acc, sep=',')
    # print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')


def showsave_plot():
    """Show or save plot."""
    if ARGS.outf:
        plt.savefig(ARGS.outf, bbox_inches='tight')
    else:
```

```
        plt.show()


def eval_nstep():
    """Evaluate model on nstep deduction."""
    # Evaluate model on every nstep test data
    results = list()
    for i in range(1, 33):
                dgen  =  LogicSeq.from_file("data/test_nstep{}.txt".format(i),  ARGS.batch_size,  pad=ARGS.pad,
verbose=False)
        model = create_model(iterations=max(ARGS.iterations, i + 1), training=True)
        results.append(model.evaluate_generator(dgen)[1])
    training, _, run = MODEL_FNAME.split('_')
    print(training, ARGS.model, ARGS.dim, run, *results, sep=',')


def plot_nstep():
    """Plot nstep results."""
    # Plot the results
    df = pd.read_csv("nstep_results.csv")
    # Get maximum run based on mean
    df['Mean'] = df.iloc[:, 4:].mean(axis=1)
    idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
    df['NStep'] = df['NStep'].astype(int)
    df = df[(df['Dim'] == ARGS.dim)]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
    plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(1, 32)
    plt.xlabel("# of steps")
    showsave_plot()


def eval_len(item='pl'):
    """Evaluate model on increasing constant and predicate lengths."""
    # Evaluate model on increasing length test data
    model = create_model(iterations=ARGS.iterations, training=True)
    training, _, run = MODEL_FNAME.split('_')
    for s in ['pl', 'cl']:
        results = list()
        for i in range(2, 65):
            dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                      ARGS.batch_size, pad=ARGS.pad, verbose=False)
            results.append(model.evaluate_generator(dgen)[1])
        print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')
```

```
def plot_len():
    """Plot increasing length results."""
    # Plot the results
    df = pd.read_csv("len_results.csv")
    df['Mean'] = df.iloc[:, 5:].mean(axis=1)
    # Get maximum run based on mean
    idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
    df = df.loc[idx]
    df = df.drop(columns=['Mean'])
    df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
    df['Len'] = df['Len'].astype(int)
    df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
    print(df.head())
    # Create plot
    sns.set_style('whitegrid')
    sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
    plt.ylim(0.4, 1.0)
    plt.ylabel("Accuracy")
    plt.xlim(2, 64)
    plt.xlabel("Length of symbols")
    showsave_plot()


def plot_dim():
    """Plot increasing dimension over increasing difficulty."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
    print(df.head())
    sns.set_style('whitegrid')
    sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
                 hue_order=['validation', 'easy', 'medium', 'hard'])
    plt.ylim(0.5, 1.0)
    plt.ylabel("Mean Accuracy")
    plt.xlim(32, 128)
    plt.xlabel("Dimension")
    plt.legend(loc='upper left')
    showsave_plot()


def plot_training():
    """Plot training method on mean accuracy per test set."""
    df = pd.read_csv("results.csv")
    df = df[(df['Model'] == ARGS.model)]
    print(df.head())
    sns.set_style('whitegrid')
    sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
    plt.ylabel("Mean Accuracy")
    plt.ylim(0.5, 1.0)
    showsave_plot()


def get_pca(context, model, dims=2):
    """Plot the PCA of predicate embeddings."""
    dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
```

```
                    train=False, shuffle=False, zeropad=False)
    embds = model.predict_generator(dgen)
    embds = embds.squeeze()
    pca = PCA(dims)
    embds = pca.fit_transform(embds)
    print("TRANSFORMED:", embds)
    print("VAR:", pca.explained_variance_ratio_)
    return embds


def offset(x):
    """Calculate offset for annotation."""
    r = np.random.randint(10, 30)
    return -r if x > 0 else r


def plot_single_preds():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    syms = "abcdefghijklmnopqrstuvwxyz"
    ctx, splits = list(), list()
    preds = list("pqrv")
    preds.extend([''.join([e] * 2) for e in preds])
    for p in preds:
        for c in syms:
            ctx.append("{}({}).".format(p, c))
        splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=len(preds))
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, "fdgm"):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_single_word():
    """Plot embeddings of single word predicates."""
    model = create_model(pca=True)
    # syms = "abcdefghijklmnopqrstuvwxyz"
    syms = WORD_INDEX.keys()
    ctx, splits = list(), list()
    # preds = list("pqrv")
    # preds.extend([''.join([e]*2) for e in preds])
    for i in WORD_INDEX.keys():
        ctx.append(i)
        splits.append(len(ctx))
    # for p in preds:
    #   for c in syms:
```

```
    #     ctx.append("{}({}).".format(p,c))
    #   splits.append(len(ctx))
    embds = get_pca(ctx, model, dims=64)
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    prev_sp = 0
    for sp in splits:
        x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
        ax.scatter(x, y, z, depthshade=False)
        for i in map(syms.index, ["blue", "someone", "are", "bob"]):
            ax.text(x[i], y[i], z[i], ctx[prev_sp + i])
        prev_sp = sp
    showsave_plot()


def plot_pred_saturation():
    """Plot predicate embedding saturation."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for i in range(2, 65):
        p = "".join(["p"] * i)
        ctx.append("{}(a).".format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    plt.scatter(embds[::2, 0], embds[::2, 1])
    plt.scatter(embds[1::2, 0], embds[1::2, 1])
    prev_sp = 0
    for i, sp in enumerate(splits):
        pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
        count = pred.count('p')
        if count <= 6:
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        elif i % 3 == 0 and i < 50 or i == len(splits) - 1 or i == len(splits) - 2:
            pred = str(count) + "*p(a)"
            xf, yf = offset(x), offset(y)
                            plt.annotate(pred,  xy=(x,  y),  xytext=(xf,  yf),  textcoords='offset  points',
arrowprops={'arrowstyle': '-'})
        prev_sp = sp
    # Plot contour
    plt.xlim(-2, 2)
    xmin, xmax = plt.xlim()
    X = np.linspace(xmin, xmax, 40)
    ymin, ymax = plt.ylim()
    Y = np.linspace(ymin, ymax, 40)
    X, Y = np.meshgrid(X, Y)
    Z = np.sqrt((X - embds[-1, 0]) ** 2 + (Y - embds[-1, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    Z = np.sqrt((X - embds[-2, 0]) ** 2 + (Y - embds[-2, 1]) ** 2)
    plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
    showsave_plot()
```

```
def plot_template(preds, temps):
    """Plot PCA of templates with given predicates."""
    model = create_model(pca=True)
    ctx, splits = list(), list()
    for t in temps:
        for p in preds:
            ctx.append(t.format(p))
        splits.append(len(ctx))
    embds = get_pca(ctx, model)
    prev_sp = 0
    for sp in splits:
        plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
        prev_sp = sp
        pred, x, y = ctx[sp - 1], embds[sp - 1, 0], embds[sp - 1, 1]
        xf, yf = offset(x), offset(y)
          plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    showsave_plot()


def plot_struct_preds():
    """Plot embeddings of different structural predicates."""
    ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
             "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
    plot_template(ps, temps)


def plot_rules():
    """Plot embeddings of rules."""
    ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
    temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
             "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
             "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
             "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
             "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
    plot_template(ps, temps)


def plot_attention():
  """Plot attention vector over given context."""
  model = create_model(iterations=ARGS.iterations, training=False)
   ctxs = ["Water is not located in forest. Forest is located in earth. Earth is located in tree. Tree is
located in surface of earth."]
  ctxs = [s.lower().replace(",","") for s in ctxs]
  q = "Earth is not located in desk."
  q = q .lower()
  q = q.replace(".","")
  q_list = [q]
  fig, axes = plt.subplots(1,1)
  for i, ctx in enumerate(ctxs):
```

```
    for j, t in enumerate(q_list):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
      out = model.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      #sims = sims.reshape(36,1)
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes.get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1, ARGS.iterations+1),
                  linewidths=0.5, square=True, cbar=False, ax=axes)
      axes.set_xlabel(q+" "+str(out) if j % 2 == 0 else "p(b) "+str(out))
  #plt.tight_layout()
  #showsave_plot()
  plt.savefig('Depth-0_CWA_case_r.png', bbox_inches='tight')


def plot_dual_attention():
    """Plot dual attention vectors of 2 models over given context."""
    modelbw = build_model("imasm", ARGS.model_dir + "curr_imasm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    modelfw = build_model("fwimarsm", ARGS.model_dir + "curr_fwimarsm64.h5",
                          char_size=len(CHAR_IDX) + 1,
                          dim=ARGS.dim, iterations=ARGS.iterations,
                          training=False)
    ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
            "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
            "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
    fig, axes = plt.subplots(1, 6)
    # Plot the attention
    for i, ctx in enumerate(ctxs):
        for j, m in enumerate([modelbw, modelfw]):
            rs = ctx.split('.')[:-1]
            dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
            out = m.predict_generator(dgen)
            sims = out[:-1]
            out = np.round(np.asscalar(out[-1]), 2)
            sims = np.stack(sims, axis=0).squeeze()
            sims = sims.T
            ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
            axes[i * 2 + j].get_xaxis().set_ticks_position('top')
            sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                        xticklabels=range(1, 5), linewidths=0.5, square=True, cbar=False, ax=axes[i * 2 + j])
            # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
            axes[i * 2 + j].set_xlabel("backward" if j % 2 == 0 else "forward")
    plt.tight_layout()
    showsave_plot()
```

```
if __name__ == '__main__':
    globals()[ARGS.function]()

==== eval_pararule.py ====
"""Evaluation module for logic-memnn"""
import argparse
from glob import glob
import numpy as np
import pandas as pd
import keras.callbacks as C
from sklearn.decomposition import PCA

from data_gen import CHAR_IDX
from utils_pararule import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS

# Arguments
parser = argparse.ArgumentParser(description="Evaluate logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-f", "--function", default="evaluate", help="Function to run.")
parser.add_argument("--outf", help="Plot to output file instead of rendering.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Evaluation batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()

if ARGS.outf:
  import matplotlib
  matplotlib.use("Agg") # Bypass X server
import matplotlib.pyplot as plt
import seaborn as sns

MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)

def create_model(**kwargs):
  """Create model from global arguments."""
  # Load in the model
  model = build_model(MODEL_NAME, MODEL_WF,
                      char_size=len(CHAR_IDX)+1,
                      dim=ARGS.dim,
                      **kwargs)
  if ARGS.summary:
    model.summary()
  return model
```

```
def evaluate():
  """Evaluate model on each test data."""
  model = create_model(iterations=ARGS.iterations, training=True)
  # training, run, glove, pararule, num = MODEL_FNAME.split('_')
  training, run, glove, pararule = MODEL_FNAME.split('_')

            for      s      in      ['test_depth_1.jsonl',      'test_depth_2.jsonl',      'test_depth_3.jsonl',
'test_depth_3ext.jsonl','test_depth_3ext_NatLang.jsonl','test_depth_5.jsonl','test_NatLang.jsonl']:
  # for s in ['val_task0.jsonl', 'val_task1.jsonl', 'val_task2.jsonl', 'val_task3.jsonl',
  #           'val_task4.jsonl', 'val_task5.jsonl', 'val_task6.jsonl', 'val_task7.jsonl']:
    results = list()
    dgen = LogicSeq.from_file("data/pararule/test/"+s, ARGS.batch_size, pad=ARGS.pad, verbose=False)
    _, acc = model.evaluate_generator(dgen) # [loss, acc]
    results.append(acc)
    print(training, ARGS.model, ARGS.dim, s, run, acc, sep=',')
  #print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')

def showsave_plot():
  """Show or save plot."""
  if ARGS.outf:
    plt.savefig(ARGS.outf, bbox_inches='tight')
  else:
    plt.show()

def eval_nstep():
  """Evaluate model on nstep deduction."""
  # Evaluate model on every nstep test data
  results = list()
  for i in range(1, 33):
    dgen = LogicSeq.from_file("data/test_nstep{}.txt".format(i), ARGS.batch_size, pad=ARGS.pad, verbose=False)
    model = create_model(iterations=max(ARGS.iterations, i+1), training=True)
    results.append(model.evaluate_generator(dgen)[1])
  training, _, run = MODEL_FNAME.split('_')
  print(training, ARGS.model, ARGS.dim, run, *results, sep=',')

def plot_nstep():
  """Plot nstep results."""
  # Plot the results
  df = pd.read_csv("nstep_results.csv")
  # Get maximum run based on mean
  df['Mean'] = df.iloc[:,4:].mean(axis=1)
  idx = df.groupby(['Model', 'Dim'])['Mean'].idxmax()
  df = df.loc[idx]
  df = df.drop(columns=['Mean'])
  df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Run'], var_name='NStep', value_name='Acc')
  df['NStep'] = df['NStep'].astype(int)
  df = df[(df['Dim'] == ARGS.dim)]
  print(df.head())
  # Create plot
  sns.set_style('whitegrid')
  sns.lineplot(x='NStep', y='Acc', hue='Model', data=df, sort=True)
  plt.vlines(3, 0.4, 1.0, colors='grey', linestyles='dashed', label='training')
  plt.ylim(0.4, 1.0)
```

```
  plt.ylabel("Accuracy")
  plt.xlim(1, 32)
  plt.xlabel("# of steps")
  showsave_plot()

def eval_len(item='pl'):
  """Evaluate model on increasing constant and predicate lengths."""
  # Evaluate model on increasing length test data
  model = create_model(iterations=ARGS.iterations, training=True)
  training, _, run = MODEL_FNAME.split('_')
  for s in ['pl', 'cl']:
    results = list()
    for i in range(2, 65):
      dgen = LogicSeq.from_file("data/test_{}{}.txt".format(s, i),
                                ARGS.batch_size, pad=ARGS.pad, verbose=False)
      results.append(model.evaluate_generator(dgen)[1])
    print(training, ARGS.model, ARGS.dim, s, run, *results, sep=',')

def plot_len():
  """Plot increasing length results."""
  # Plot the results
  df = pd.read_csv("len_results.csv")
  df['Mean'] = df.iloc[:,5:].mean(axis=1)
  # Get maximum run based on mean
  idx = df.groupby(['Model', 'Dim', 'Symbol'])['Mean'].idxmax()
  df = df.loc[idx]
  df = df.drop(columns=['Mean'])
  df = pd.melt(df, id_vars=['Training', 'Model', 'Dim', 'Symbol', 'Run'], var_name='Len', value_name='Acc')
  df['Len'] = df['Len'].astype(int)
  df = df[(df['Dim'] == ARGS.dim) & (~df['Model'].isin(['imasm', 'imarsm']))]
  print(df.head())
  # Create plot
  sns.set_style('whitegrid')
  sns.lineplot(x='Len', y='Acc', hue='Model', style='Symbol', data=df, sort=True)
  plt.ylim(0.4, 1.0)
  plt.ylabel("Accuracy")
  plt.xlim(2, 64)
  plt.xlabel("Length of symbols")
  showsave_plot()

def plot_dim():
  """Plot increasing dimension over increasing difficulty."""
  df = pd.read_csv("results.csv")
  df = df[(df['Model'] == ARGS.model) & (df['Training'] == 'curr')]
  print(df.head())
  sns.set_style('whitegrid')
  sns.lineplot(x='Dim', y='Mean', hue='Set', data=df,
               hue_order=['validation', 'easy', 'medium', 'hard'])
  plt.ylim(0.5, 1.0)
  plt.ylabel("Mean Accuracy")
  plt.xlim(32, 128)
  plt.xlabel("Dimension")
  plt.legend(loc='upper left')
  showsave_plot()
```

```
def plot_training():
  """Plot training method on mean accuracy per test set."""
  df = pd.read_csv("results.csv")
  df = df[(df['Model'] == ARGS.model)]
  print(df.head())
  sns.set_style('whitegrid')
  sns.barplot(x='Training', y='Mean', hue='Set', errwidth=1.2, capsize=0.025, data=df)
  plt.ylabel("Mean Accuracy")
  plt.ylim(0.5, 1.0)
  showsave_plot()

def get_pca(context, model, dims=2):
  """Plot the PCA of predicate embeddings."""
  dgen = LogicSeq([[(context, "z(z).", 0)]], 1,
                  train=False, shuffle=False, zeropad=False)
  embds = model.predict_generator(dgen)
  embds = embds.squeeze()
  pca = PCA(dims)
  embds = pca.fit_transform(embds)
  print("TRANSFORMED:", embds)
  print("VAR:", pca.explained_variance_ratio_)
  return embds

def offset(x):
  """Calculate offset for annotation."""
  r = np.random.randint(10, 30)
  return -r if x > 0 else r

def plot_single_preds():
  """Plot embeddings of single word predicates."""
  model = create_model(pca=True)
  syms = "abcdefghijklmnopqrstuvwxyz"
  ctx, splits = list(), list()
  preds = list("pqrv")
  preds.extend(['' .join([e]*2) for e in preds])
  for p in preds:
    for c in syms:
      ctx.append("{}({}).".format(p,c))
    splits.append(len(ctx))
  embds = get_pca(ctx, model, dims=len(preds))
  from mpl_toolkits.mplot3d import Axes3D
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  prev_sp = 0
  for sp in splits:
    x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
    ax.scatter(x, y, z, depthshade=False)
    for i in map(syms.index, "fdgm"):
      ax.text(x[i], y[i], z[i], ctx[prev_sp+i])
    prev_sp = sp
  showsave_plot()

def plot_single_word():
```