Lab 4: Data Imputation using an Autoencoder

In this lab, you will build and train an autoencoder to impute (or "fill in") missing data.

We will be using the Adult Data Set provided by the UCI Machine Learning Repository [1], available at https://archive.ics.uci.edu/ml/datasets/adult. The data set contains census record files of adults, including their age, martial status, the type of work they do, and other features.

Normally, people use this data set to build a supervised classification model to classify whether a person is a high income earner. We will not use the dataset for this original intended purpose.

Instead, we will perform the task of imputing (or "filling in") missing values in the dataset. For example, we may be missing one person's martial status, and another person's age, and a third person's level of education. Our model will predict the missing features based on the information that we do have about each person.

We will use a variation of a denoising autoencoder to solve this data imputation problem. Our autoencoder will be trained using inputs that have one categorical feature artificially removed, and the goal of the autoencoder is to correctly reconstruct all features, including the one removed from the input.

In the process, you are expected to learn to:

1. Clean and process continuous and categorical data for machine learning.
2. Implement an autoencoder that takes continuous and categorical (one-hot) inputs.
3. Tune the hyperparameters of an autoencoder.
4. Use baseline models to help interpret model performance.

[1] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

What to submit

Submit a PDF file containing all your code, outputs, and write-up. You can produce a PDF of your Google Colab file by going to File > Print and then save as PDF. The Colab instructions have more information.

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Colab Link

Include a link to your Colab file here. If you would like the TA to look at your Colab file in case your solutions are cut off, **please make sure that your Colab file is publicly accessible at the time of submission**.

Colab Link: https://colab.research.google.com/drive/1LAvhPCepcjKBV3SG3S5KpFlLiDBoeGEl?usp=sharing

```
import csv
import numpy as np
import random
import torch
import torch.utils.data
```

Part 0

We will be using a package called `pandas` for this assignment.

If you are using Colab, `pandas` should already be available. If you are using your own computer, installation instructions for `pandas` are available here: https://pandas.pydata.org/pandas-docs/stable/install.html

```
import pandas as pd
```

Part 1. Data Cleaning [15 pt]

The adult.data file is available at `https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data`

The function `pd.read_csv` loads the adult.data file into a pandas dataframe. You can read about the pandas documentation for `pd.read_csv` at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

```
header = ['age', 'work', 'fnlwgt', 'edu', 'yredu', 'marriage',
'occupation',
 'relationship', 'race', 'sex', 'capgain', 'caploss', 'workhr',
'country']
df = pd.read_csv(

"https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult
.data",
    names=header,
    index_col=False)
```

```
<ipython-input-2-037957db2593>:3: ParserWarning: Length of header or
names does not match length of data. This leads to a loss of data with
index_col=False.
  df = pd.read_csv(
```

```
df.shape # there are 32561 rows (records) in the data frame, and 14
columns (features)
```

```
(32561, 14)
```

Part (a) Continuous Features [3 pt]

For each of the columns `["age", "yredu", "capgain", "caploss", "workhr"]`, report the minimum, maximum, and average value across the dataset.

Then, normalize each of the features `["age", "yredu", "capgain", "caploss", "workhr"]` so that their values are always between 0 and 1. Make sure that you are actually modifying the dataframe `df`.

Like numpy arrays and torch tensors, pandas data frames can be sliced. For example, we can display the first 3 rows of the data frame (3 records) below.

```
df[:3] # show the first 3 records
newdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
print(newdf.max())
print(newdf.min())
print(newdf.mean())

df["age"] = df["age"] / 90
df["yredu"] = df["yredu"] / 16
df["capgain"] = df["capgain"] / 99999
df["caploss"] = df["caploss"] / 4356
df["workhr"] = df["workhr"] / 99
df[:3]
```

```
age        1.0
yredu      1.0
capgain    1.0
caploss    1.0
workhr     1.0
dtype: float64
age        0.188889
yredu      0.062500
capgain    0.000000
caploss    0.000000
workhr     0.010101
dtype: float64
age        0.428685
yredu      0.630042
capgain    0.010777
caploss    0.020042
workhr     0.408459
dtype: float64
```


```
{"summary":"{\n  \"name\": \"df[:3]\",\n  \"rows\": 3,\n  \"fields\":
[\n    {\n      \"column\": \"age\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.0008220158170962216,\n
\"min\": 0.004691358024691358,\n        \"max\": 0.00617283950617284,\
n        \"num_unique_values\": 3,\n        \"samples\": [\n
0.004814814814814815,\n          0.00617283950617284,\n
```

```
0.004691358024691358\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"work\",\n      \"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 3,\n        \"samples\": [\n          \" State-
gov\",\n          \" Self-emp-not-inc\",\n          \" Private\"\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"fnlwgt\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\": 78130,\n
\"min\": 77516,\n        \"max\": 215646,\n
\"num_unique_values\": 3,\n        \"samples\": [\n          77516,\n
83311,\n          215646\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"edu\",\n      \"properties\": {\n        \"dtype\":
\"string\",\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          \" HS-grad\",\n          \" Bachelors\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"yredu\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\":
0.009021097956087904,\n        \"min\": 0.03515625,\n        \"max\":
0.05078125,\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          0.03515625,\n          0.05078125\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"marriage\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\": 3,\n
\"samples\": [\n          \" Never-married\",\n          \" Married-
civ-spouse\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"occupation\",\n      \"properties\": {\n        \"dtype\":
\"string\",\n        \"num_unique_values\": 3,\n        \"samples\":
[\n          \" Adm-clerical\",\n          \" Exec-managerial\"\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"relationship\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 2,\n        \"samples\": [\n          \"
Husband\",\n          \" Not-in-family\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"race\",\n      \"properties\": {\n
\"dtype\": \"category\",\n        \"num_unique_values\": 1,\n
\"samples\": [\n          \" White\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"sex\",\n      \"properties\": {\n
\"dtype\": \"category\",\n        \"num_unique_values\": 1,\n
\"samples\": [\n          \" Male\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"capgain\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
1.2551845887845037e-07,\n        \"min\": 0.0,\n        \"max\":
2.1740434806522087e-07,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          0.0\n        ],\n        \"semantic_type\":
```

```
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"caploss\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.0,\n        \"min\": 0.0,\n
\"max\": 0.0,\n        \"num_unique_values\": 1,\n        \"samples\":
[\n          0.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"workhr\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 0.001590496609337812,\n        \"min\":
0.0013263952657892053,\n        \"max\": 0.004081216202428324,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
0.0013263952657892053\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"country\",\n      \"properties\": {\n        \"dtype\":
\"category\",\n        \"num_unique_values\": 1,\n        \"samples\":
[\n          \" United-States\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe"}
```


Alternatively, we can slice based on column names, for example `df["race"]`, `df["hr"]`, or even index multiple columns like below.

```
subdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
subdf[:3] # show the first 3 records
```


```
{"summary":"{\n  \"name\": \"subdf[:3] # show the first 3 records\",\n
\"rows\": 3,\n  \"fields\": [\n    {\n      \"column\": \"age\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.0008220158170962216,\n        \"min\": 0.004691358024691358,\n
\"max\": 0.00617283950617284,\n        \"num_unique_values\": 3,\n
\"samples\": [\n          0.004814814814814815,\n
0.00617283950617284,\n          0.004691358024691358\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"yredu\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\":
0.009021097956087904,\n        \"min\": 0.03515625,\n        \"max\":
0.05078125,\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          0.03515625,\n          0.05078125\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"capgain\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
1.2551845887845037e-07,\n        \"min\": 0.0,\n        \"max\":
2.1740434806522087e-07,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          0.0,\n          2.1740434806522087e-07\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"caploss\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.0,\n        \"min\": 0.0,\n        \"max\": 0.0,\n
\"num_unique_values\": 1,\n        \"samples\": [\n          0.0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
```

```
}\n    },\n    {\n      \"column\": \"workhr\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
0.0015904966093378 12,\n        \"min\": 0.0013263952657892053,\n
\"max\": 0.004081216202428324,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          0.0013263952657892053\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe"}
```


Numpy works nicely with pandas, like below:

```
np.sum(subdf["caploss"])
```

```
0.14981499610850976
```

Just like numpy arrays, you can modify entire columns of data rather than one scalar element at a time. For example, the code

```
df["age"] = df["age"] + 1
```

would increment everyone's age by 1.

```

```

Part (b) Categorical Features [1 pt]

What percentage of people in our data set are male? Note that the data labels all have an unfortunate space in the beginning, e.g. " Male" instead of "Male".

What percentage of people in our data set are female?

```
# hint: you can do something like this in pandas
sum(df["sex"] == " Male")
sum(df["sex"] == " Female")
percentage_female = sum(df["sex"] == " Female") / (sum(df["sex"] == "
Male") + sum(df["sex"] == " Female")) * 100
print(percentage_female, "%")
```

```
33.07945087681583 %
```

Part (c) [2 pt]

Before proceeding, we will modify our data frame in a couple more ways:

1. We will restrict ourselves to using a subset of the features (to simplify our autoencoder)
2. We will remove any records (rows) already containing missing values, and store them in a second dataframe. We will only use records without missing values to train our autoencoder.

Both of these steps are done for you, below.

How many records contained missing features? What percentage of records were removed?

```
contcols = ["age", "yredu", "capgain", "caploss", "workhr"]
catcols = ["work", "marriage", "occupation", "edu", "relationship",
"sex"]
features = contcols + catcols
df = df[features]

missing = pd.concat([df[c] == " ?" for c in catcols],
axis=1).any(axis=1)
df_with_missing = df[missing]
df_not_missing = df[~missing]
```

Part (d) One-Hot Encoding [1 pt]

What are all the possible values of the feature "work" in `df_not_missing`? You may find the Python function `set` useful.

```
workset = set(df_not_missing["work"])
print(workset)
```

```
{' Self-emp-inc', ' Federal-gov', ' Local-gov', ' Without-pay', '
State-gov', ' Private', ' Self-emp-not-inc'}
```

We will be using a one-hot encoding to represent each of the categorical variables. Our autoencoder will be trained using these one-hot encodings.

We will use the pandas function `get_dummies` to produce one-hot encodings for all of the categorical variables in `df_not_missing`.

```
data = pd.get_dummies(df_not_missing)

data[:3]
```

```
{"type":"dataframe"}
```

Part (e) One-Hot Encoding [2 pt]

The dataframe `data` contains the cleaned and normalized data that we will use to train our denoising autoencoder.

How many **columns** (features) are in the dataframe `data`?

Briefly explain where that number come from.

```
"As shown in the previous output, there are 57 columns. This is
because all the possible values of the categorical data columns became
their own columns such that the new columns only contains boolean
values."
```

```
{"type":"string"}
```

Part (f) One-Hot Conversion [3 pt]

We will convert the pandas data frame `data` into numpy, so that it can be further converted into a PyTorch tensor. However, in doing so, we lose the column label information that a panda data frame automatically stores.

Complete the function `get_categorical_value` that will return the named value of a feature given a one-hot embedding. You may find the global variables `cat_index` and `cat_values` useful. (Display them and figure out what they are first.)

We will need this function in the next part of the lab to interpret our autoencoder outputs. So, the input to our function `get_categorical_values` might not actually be "one-hot" -- the input may instead contain real-valued predictions from our neural network.

```
datanp = data.values.astype(np.float32)

cat_index = {}  # Mapping of feature -> start index of feature in a
record
cat_values = {} # Mapping of feature -> list of categorical values the
feature can take

# build up the cat_index and cat_values dictionary
for i, header in enumerate(data.keys()):
    if "_" in header: # categorical header
        feature, value = header.split()
        feature = feature[:-1] # remove the last char; it is always an
underscore
        if feature not in cat_index:
            cat_index[feature] = i
            cat_values[feature] = [value]
        else:
            cat_values[feature].append(value)

print(cat_index)
print(cat_values)

def get_onehot(record, feature):
    """
    Return the portion of `record` that is the one-hot encoding
    of `feature`. For example, since the feature "work" is stored
    in the indices [5:12] in each record, calling `get_range(record,
"work")`
    is equivalent to accessing `record[5:12]`.

    Args:
        - record: a numpy array representing one record, formatted
                  the same way as a row in `data.np`
        - feature: a string, should be an element of `catcols`
```

```
    """
    start_index = cat_index[feature]
    stop_index = cat_index[feature] + len(cat_values[feature])
    return record[start_index:stop_index]

def get_categorical_value(onehot, feature):
    """
    Return the categorical value name of a feature given
    a one-hot vector representing the feature.

    Args:
        - onehot: a numpy array one-hot representation of the feature
        - feature: a string, should be an element of `catcols`

    Examples:

    >>> get_categorical_value(np.array([0., 0., 0., 0., 0., 1., 0.]),
"work")
    'State-gov'
    >>> get_categorical_value(np.array([0.1, 0., 1.1, 0.2, 0., 1.,
0.]), "work")
    'Private'
    """
    # <----- TODO: WRITE YOUR CODE HERE ----->
    # You may find the variables `cat_index` and `cat_values`
    # (created above) useful.

    if feature == "work":
      return cat_values["work"][np.argmax(onehot)]
    elif feature == "marriage":
      return cat_values["marriage"][np.argmax(onehot) +
cat_index["marriage"]]
    elif feature == "occupation":
      return cat_values["occupation"][np.argmax(onehot) +
cat_index["occupation"]]
    elif feature == "edu":
      return cat_values["edu"][np.argmax(onehot) + cat_index["edu"]]
    elif feature == "relationship":
      return cat_values["relationship"][np.argmax(onehot) +
cat_index["relationship"]]
    elif feature == "sex":
      return cat_values["sex"][np.argmax(onehot) + cat_index["sex"]]

{'work': 5, 'marriage': 12, 'occupation': 19, 'edu': 33,
'relationship': 49, 'sex': 55}
{'work': ['Federal-gov', 'Local-gov', 'Private', 'Self-emp-inc',
'Self-emp-not-inc', 'State-gov', 'Without-pay'], 'marriage':
['Divorced', 'Married-AF-spouse', 'Married-civ-spouse', 'Married-
spouse-absent', 'Never-married', 'Separated', 'Widowed'],
'occupation': ['Adm-clerical', 'Armed-Forces', 'Craft-repair', 'Exec-
```

```
managerial', 'Farming-fishing', 'Handlers-cleaners', 'Machine-op-
inspct', 'Other-service', 'Priv-house-serv', 'Prof-specialty',
'Protective-serv', 'Sales', 'Tech-support', 'Transport-moving'],
'edu': ['10th', '11th', '12th', '1st-4th', '5th-6th', '7th-8th',
'9th', 'Assoc-acdm', 'Assoc-voc', 'Bachelors', 'Doctorate', 'HS-grad',
'Masters', 'Preschool', 'Prof-school', 'Some-college'],
'relationship': ['Husband', 'Not-in-family', 'Other-relative', 'Own-
child', 'Unmarried', 'Wife'], 'sex': ['Female', 'Male']}

# more useful code, used during training, that depends on the function
# you write above

def get_feature(record, feature):
    """
    Return the categorical feature value of a record
    """
    onehot = get_onehot(record, feature)
    return get_categorical_value(onehot, feature)

def get_features(record):
    """
    Return a dictionary of all categorical feature values of a record
    """
    return { f: get_feature(record, f) for f in catcols }
```

Part (g) Train/Test Split [3 pt]

Randomly split the data into approximately 70% training, 15% validation and 15% test.

Report the number of items in your training, validation, and test set.

```
# set the numpy seed for reproducibility
#
https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed
.html
np.random.seed(50)

# todo
np.random.shuffle(datanp)

#training data
training_set = datanp[:int(0.7*len(datanp))]
training_loader = torch.utils.data.DataLoader(training_set,
batch_size=64, shuffle=True)

#validation data
validation_set = datanp[int(0.7*len(datanp)):int(0.85*len(datanp))]
validation_loader = torch.utils.data.DataLoader(validation_set,
batch_size=64, shuffle=True)
```

```
#test data
test_set = datanp[int(0.85*len(datanp)):]
test_loader = torch.utils.data.DataLoader(test_set, batch_size=64,
shuffle=True)

print(np.shape(training_set))
print(np.shape(validation_set))
print(np.shape(test_set))
```

```
(21502, 57)
(4608, 57)
(4608, 57)
```

Part 2. Model Setup [5 pt]

Part (a) [4 pt]

Design a fully-connected autoencoder by modifying the `encoder` and `decoder` below.

The input to this autoencoder will be the features of the `data`, with one categorical feature recorded as "missing". The output of the autoencoder should be the reconstruction of the same features, but with the missing value filled in.

Note: Do not reduce the dimensionality of the input too much! The output of your embedding is expected to contain information about ~11 features.

```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.name = "Autoencoder"
        self.encoder = nn.Sequential(
            nn.Linear(57, 57), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(57,57),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(57, 57), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(57,57),
            nn.ReLU(),
            nn.Sigmoid() # get to the range (0, 1)
        )

    def forward(self, x):
```

```
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Part (b) [1 pt]

Explain why there is a sigmoid activation in the last step of the decoder.

(**Note**: the values inside the data frame `data` and the training code in Part 3 might be helpful.)

```
"We use sigmoid since we want the output data to fall within the range
of [0,1], since this is the range we used to normalize the continuous
data, and also because we used onehot to convert the categorical
values into boolean values of 1s and 0s"
"The NN might learn that all the data falls with [0,1] without
sigmoid, but this is just to make the learning process more stable"
```

```
{"type":"string"}
```

Part 3. Training [18]

Part (a) [6 pt]

We will train our autoencoder in the following way:

- In each iteration, we will hide one of the categorical features using the `zero_out_random_features` function
- We will pass the data with one missing feature through the autoencoder, and obtain a reconstruction
- We will check how close the reconstruction is compared to the original data -- including the value of the missing feature

Complete the code to train the autoencoder, and plot the training and validation loss every few iterations. You may also want to plot training and validation "accuracy" every few iterations, as we will define in part (b). You may also want to checkpoint your model every few iterations or epochs.

Use `nn.MSELoss()` as your loss function. (Side note: you might recognize that this loss function is not ideal for this problem, but we will use it anyway.)

```
def zero_out_feature(records, feature):
    """ Set the feature missing in records, by setting the appropriate
    columns of records to 0
    """
    start_index = cat_index[feature]
    stop_index = cat_index[feature] + len(cat_values[feature])
    records[:, start_index:stop_index] = 0
    return records
```

```
def zero_out_random_feature(records):
    """ Set one random feature missing in records, by setting the
    appropriate columns of records to 0
    """
    return zero_out_feature(records, random.choice(catcols))

def get_model_name(name, batch_size, learning_rate, epoch):
    """ Generate a name for the model consisting of all the
hyperparameter values

    Args:
        config: Configuration object containing the hyperparameters
    Returns:
        path: A string with the hyperparameter name and value
concatenated
    """
    path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(name,
                                                   batch_size,
                                                   learning_rate,
                                                   epoch)
    return path

def plot_training_curve(path):
    """ Plots the training curve for a model run, given the csv files
    containing the train/validation error/loss.

    Args:
        path: The base path of the csv files produced during training
    """
    import matplotlib.pyplot as plt
    train_loss = np.loadtxt("{}_train_loss.csv".format(path))
    val_loss = np.loadtxt("{}_val_loss.csv".format(path))
    accuracy = np.loadtxt("{}_accuracy.csv".format(path))
    plt.title("Train vs Validation Loss")
    n = len(train_loss)
    plt.plot(range(1,n+1), train_loss, label="Train")
    plt.plot(range(1,n+1), val_loss, label="Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()
    plt.title("Accuracy")
    plt.plot(range(1,n+1), accuracy, label="Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

def train(model, train_loader, valid_loader, num_epochs=50,
learning_rate=1e-4):
```

```
    import time
    """ Training loop. You should update this."""
    start_time = time.time()
    torch.manual_seed(42)
    batch_size = 16
    training_loader = torch.utils.data.DataLoader(training_set,
batch_size, shuffle=True)
    validation_loader = torch.utils.data.DataLoader(validation_set,
batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(test_set, batch_size,
shuffle=True)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_loss = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    accuracy = np.zeros(num_epochs)

    for epoch in range(num_epochs):
        total_train_loss = 0.0
        total_val_loss = 0.0

        for i, data in enumerate(train_loader):
            datam = zero_out_random_feature(data.clone()) # zero out
one categorical feature
            recon = model(datam) #feed missing data into model to get
reconstruction
            loss = criterion(recon, data) #compute loss
            loss.backward() #every time this is called, computed grad
is added to existing grad
            optimizer.step() #update parameters based on computed grad
            optimizer.zero_grad() #old gradients may cause
unpredictable behavior
            total_train_loss += loss.item()

        train_loss[epoch] = total_train_loss / (i + 1)


        for c, dataval in enumerate(valid_loader):
            datavalm = zero_out_random_feature(dataval.clone())
            reconval = model(datavalm)
            total_val_loss += criterion(reconval, dataval).item()

        val_loss[epoch] = total_val_loss / (c + 1)

        accuracy[epoch] = get_accuracy(model, valid_loader)

        print(("Epoch {}: Train loss: {} |"+"Validation loss: {}
|"+"Accuracy: {}").format(
                      epoch + 1,
```

```
                      train_loss[epoch],
                      val_loss[epoch],
                      accuracy[epoch]))
        path = get_model_name(model.name, batch_size, learning_rate,
epoch)
        torch.save(model.state_dict(), path)

        end_time = time.time()
        elapsed_time = end_time - start_time
        print("Total time elapsed: {:.2f}
seconds".format(elapsed_time))

        epochs = np.arange(1, num_epochs + 1)
        np.savetxt("{}_train_loss.csv".format(path), train_loss)
        np.savetxt("{}_val_loss.csv".format(path), val_loss)
        np.savetxt("{}_accuracy.csv".format(path), accuracy)



    return model
```

Part (b) [3 pt]

While plotting training and validation loss is valuable, loss values are harder to compare than accuracy percentages. It would be nice to have a measure of "accuracy" in this problem.

Since we will only be imputing missing categorical values, we will define an accuracy measure. For each record and for each categorical feature, we determine whether the model can predict the categorical feature given all the other features of the record.

A function `get_accuracy` is written for you. It is up to you to figure out how to use the function. **You don't need to submit anything in this part.** To earn the marks, correctly plot the training and validation accuracy every few iterations as part of your training curve.

```
def get_accuracy(model, data_loader):
    """Return the "accuracy" of the autoencoder model across a data
set.
    That is, for each record and for each categorical feature,
    we determine whether the model can successfully predict the value
    of the categorical feature given all the other features of the
    record. The returned "accuracy" measure is the percentage of times
    that our model is successful.

    Args:
       - model: the autoencoder model, an instance of nn.Module
       - data_loader: an instance of torch.utils.data.DataLoader

    Example (to illustrate how get_accuracy is intended to be called.
             Depending on your variable naming this code might require
             modification.)
```

```
        >>> model = AutoEncoder()
        >>> vdl = torch.utils.data.DataLoader(data_valid,
batch_size=256, shuffle=True)
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            inp = item.detach().numpy()
            out = model(zero_out_feature(item.clone(),
col)).detach().numpy()
            for i in range(out.shape[0]): # record in minibatch
                acc += int(get_feature(out[i], col) ==
get_feature(inp[i], col))
                total += 1
    return acc / total
```

Part (c) [4 pt]

Run your updated training code, using reasonable initial hyperparameters.

Include your training curve in your submission.

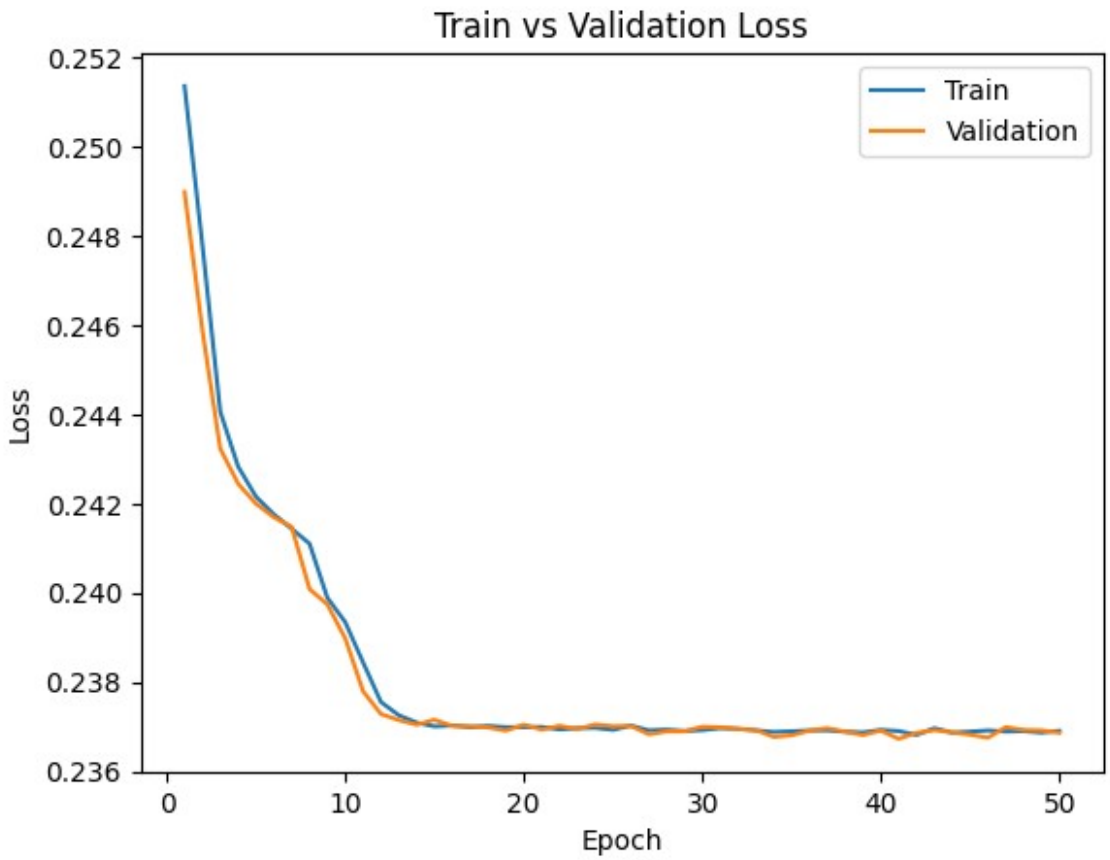
```
model = AutoEncoder()
train(model, training_loader, validation_loader)
path = get_model_name("Autoencoder", batch_size=16, learning_rate=1e-
4, epoch=49)
plot_training_curve(path)
```

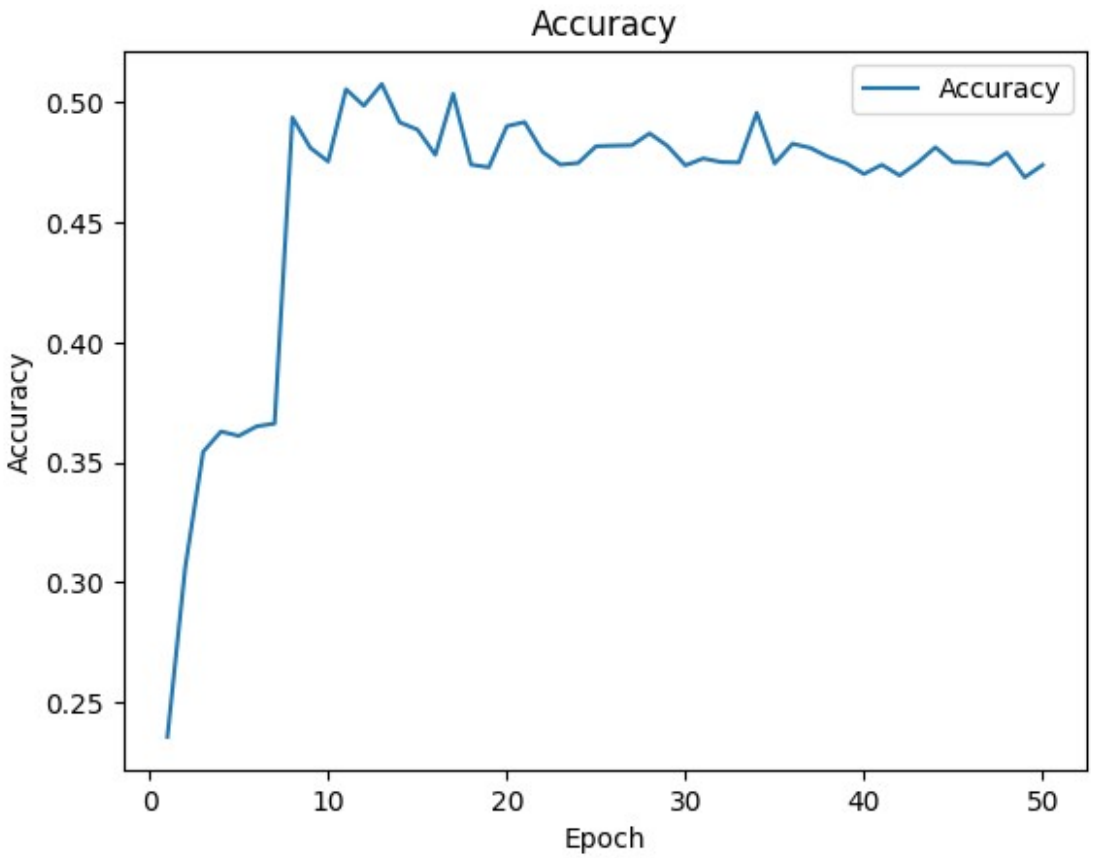
```
Epoch 1: Train loss: 0.25135337681110415 |Validation loss:
0.24897939732505214 |Accuracy: 0.2355324074074074
Total time elapsed: 1.30 seconds
Epoch 2: Train loss: 0.24778691084966772 |Validation loss:
0.24587651073104805 |Accuracy: 0.3063151041666667
Total time elapsed: 2.60 seconds
Epoch 3: Train loss: 0.24405859880858943 |Validation loss:
0.24322769128614002 |Accuracy: 0.3543836805555556
Total time elapsed: 3.85 seconds
Epoch 4: Train loss: 0.24282846985650913 |Validation loss:
0.24244424204031625 |Accuracy: 0.3628472222222222
Total time elapsed: 5.11 seconds
Epoch 5: Train loss: 0.24215451902931645 |Validation loss:
0.2419962827116251 |Accuracy: 0.36103877314814814
Total time elapsed: 6.37 seconds
Epoch 6: Train loss: 0.24175909112784125 |Validation loss:
0.24169284974535307 |Accuracy: 0.3650173611111111
Total time elapsed: 7.63 seconds
```

```
Epoch 7: Train loss: 0.2414356316218064 |Validation loss:
0.2414781887912088 |Accuracy: 0.36617476851851855
Total time elapsed: 9.38 seconds
Epoch 8: Train loss: 0.24110201333782502 |Validation loss:
0.24008438156710732 |Accuracy: 0.4938151041666667
Total time elapsed: 11.31 seconds
Epoch 9: Train loss: 0.23988771571644715 |Validation loss:
0.23973861729933155 |Accuracy: 0.48115596064814814
Total time elapsed: 12.60 seconds
Epoch 10: Train loss: 0.23934429972654298 |Validation loss:
0.2389806273082892 |Accuracy: 0.47544126157407407
Total time elapsed: 13.87 seconds
Epoch 11: Train loss: 0.23844070244757903 |Validation loss:
0.23779350664052698 |Accuracy: 0.5055700231481481
Total time elapsed: 15.15 seconds
Epoch 12: Train loss: 0.237553246895827 |Validation loss:
0.23728425035046208 |Accuracy: 0.4986979166666667
Total time elapsed: 16.44 seconds
Epoch 13: Train loss: 0.23725691628420637 |Validation loss:
0.23714834596547815 |Accuracy: 0.5077401620370371
Total time elapsed: 17.75 seconds
Epoch 14: Train loss: 0.2370916342894946 |Validation loss:
0.23704622892869842 |Accuracy: 0.4917896412037037
Total time elapsed: 19.05 seconds
Epoch 15: Train loss: 0.23700818605720997 |Validation loss:
0.2371556396699614 |Accuracy: 0.48875144675925924
Total time elapsed: 20.33 seconds
Epoch 16: Train loss: 0.23702867018679777 |Validation loss:
0.23701790037254492 |Accuracy: 0.47822627314814814
Total time elapsed: 21.81 seconds
Epoch 17: Train loss: 0.23699063448501484 |Validation loss:
0.2370117348101404 |Accuracy: 0.5036530671296297
Total time elapsed: 23.71 seconds
Epoch 18: Train loss: 0.23701831237191245 |Validation loss:
0.23698387088047135 |Accuracy: 0.47413917824074076
Total time elapsed: 25.18 seconds
Epoch 19: Train loss: 0.23699086106249265 |Validation loss:
0.2369126985884375 |Accuracy: 0.4730179398148148
Total time elapsed: 26.47 seconds
Epoch 20: Train loss: 0.236991270622682 |Validation loss:
0.23704594580663574 |Accuracy: 0.49016203703703703
Total time elapsed: 27.75 seconds
Epoch 21: Train loss: 0.23699667029792354 |Validation loss:
0.23693719796008533 |Accuracy: 0.4918619791666667
Total time elapsed: 29.06 seconds
Epoch 22: Train loss: 0.23694059536570594 |Validation loss:
0.237020927377873 |Accuracy: 0.4794921875
Total time elapsed: 30.34 seconds
Epoch 23: Train loss: 0.23696748748244273 |Validation loss:
```

```
0.23694834071728918 |Accuracy: 0.47413917824074076
Total time elapsed: 31.61 seconds
Epoch 24: Train loss: 0.23697547601269825 |Validation loss:
0.2370496535052856 |Accuracy: 0.4748263888888889
Total time elapsed: 32.93 seconds
Epoch 25: Train loss: 0.236930146325557 |Validation loss:
0.23701083660125732 |Accuracy: 0.4818070023148148
Total time elapsed: 34.23 seconds
Epoch 26: Train loss: 0.2370263801532842 |Validation loss:
0.2370176555381881 |Accuracy: 0.4820601851851852
Total time elapsed: 36.12 seconds
Epoch 27: Train loss: 0.23691912412288643 |Validation loss:
0.23683091356522507 |Accuracy: 0.4822048611111111
Total time elapsed: 37.87 seconds
Epoch 28: Train loss: 0.23693458327934855 |Validation loss:
0.2368954767783483 |Accuracy: 0.48716001157407407
Total time elapsed: 39.15 seconds
Epoch 29: Train loss: 0.23690464804392486 |Validation loss:
0.23689702567127016 |Accuracy: 0.48191550925925924
Total time elapsed: 40.47 seconds
Epoch 30: Train loss: 0.2369245154605735 |Validation loss:
0.23699885668853918 |Accuracy: 0.4738136574074074
Total time elapsed: 41.74 seconds
Epoch 31: Train loss: 0.23696753209722893 |Validation loss:
0.23698833460609117 |Accuracy: 0.47670717592592593
Total time elapsed: 43.04 seconds
Epoch 32: Train loss: 0.2369538890197873 |Validation loss:
0.23694909839994377 |Accuracy: 0.4752242476851852
Total time elapsed: 44.36 seconds
Epoch 33: Train loss: 0.23691749572753906 |Validation loss:
0.23692573027478325 |Accuracy: 0.4750434027777778
Total time elapsed: 45.65 seconds
Epoch 34: Train loss: 0.2368826103352365 |Validation loss:
0.23677252957390416 |Accuracy: 0.4957682291666667
Total time elapsed: 46.93 seconds
Epoch 35: Train loss: 0.23689751211731208 |Validation loss:
0.23680873463551202 |Accuracy: 0.474609375
Total time elapsed: 48.67 seconds
Epoch 36: Train loss: 0.23691658351925157 |Validation loss:
0.23691244257820976 |Accuracy: 0.4828559027777778
Total time elapsed: 50.63 seconds
Epoch 37: Train loss: 0.23691874937642188 |Validation loss:
0.23696709589825737 |Accuracy: 0.4811197916666667
Total time elapsed: 51.94 seconds
Epoch 38: Train loss: 0.23688928459194444 |Validation loss:
0.23689104740818342 |Accuracy: 0.4773582175925926
Total time elapsed: 53.26 seconds
Epoch 39: Train loss: 0.2368724611366079 |Validation loss:
0.23681417231758436 |Accuracy: 0.47475405092592593
```

```
Total time elapsed: 54.54 seconds
Epoch 40: Train loss: 0.2369280666822479 |Validation loss:
0.23691481455332702 |Accuracy: 0.47023292824074076
Total time elapsed: 55.78 seconds
Epoch 41: Train loss: 0.2369033751920575 |Validation loss:
0.2367289209117492 |Accuracy: 0.4740668402777778
Total time elapsed: 57.07 seconds
Epoch 42: Train loss: 0.23681752361534608 |Validation loss:
0.23686139244172308 |Accuracy: 0.4696180555555556
Total time elapsed: 58.35 seconds
Epoch 43: Train loss: 0.2369674522695797 |Validation loss:
0.2369234665400452 |Accuracy: 0.47489872685185186
Total time elapsed: 59.63 seconds
Epoch 44: Train loss: 0.23686760473286822 |Validation loss:
0.23687247931957245 |Accuracy: 0.48137297453703703
Total time elapsed: 61.15 seconds
Epoch 45: Train loss: 0.23689099818113304 |Validation loss:
0.23682199749681684 |Accuracy: 0.47511574074074076
Total time elapsed: 63.10 seconds
Epoch 46: Train loss: 0.23691659642472154 |Validation loss:
0.2367543230454127 |Accuracy: 0.4749710648148148
Total time elapsed: 64.62 seconds
Epoch 47: Train loss: 0.23689311804870763 |Validation loss:
0.23699113871488306 |Accuracy: 0.47413917824074076
Total time elapsed: 65.90 seconds
Epoch 48: Train loss: 0.2369026378063219 |Validation loss:
0.23692689111663234 |Accuracy: 0.4791304976851852
Total time elapsed: 67.19 seconds
Epoch 49: Train loss: 0.23686952445478665 |Validation loss:
0.23691857481996217 |Accuracy: 0.4688585069444444
Total time elapsed: 68.47 seconds
Epoch 50: Train loss: 0.23690901009277218 |Validation loss:
0.23686697458227476 |Accuracy: 0.4739583333333333
Total time elapsed: 69.74 seconds
```


Train vs Validation Loss
Train
Validation
Loss
Epoch
0.252
0.250
0.248
0.246
0.244
0.242
0.240
0.238
0.236
0
10
20
30
40
50



Part (d) [5 pt]

Tune your hyperparameters, training at least 4 different models (4 sets of hyperparameters).

Do not include all your training curves. Instead, explain what hyperparameters you tried, what their effect was, and what your thought process was as you chose the next set of hyperparameters to try.

```
"Baseline Result: 0.43 accuracy, 0.24 training loss, 0.24 validation
loss"

"First try: Increasing number of epochs to 50."
"Result: 0.47 accuracy, 0.237 training loss, 0.237 validation loss"
"Discussion: Increasing number of epochs drastically increased
training time, with only a marginal benefit. "

"Second try: Increasing batch size to 128. Based on previous labs,
increasing batch size may assist in a higher accuracy."
"Result: 0.398 accuracy, 0.238 training loss, 0.238 validation loss"
"Discussion: Increasing batchsize made results less accurate"

"Third try: Decrease lr to 1e-5. Since the accuracy graph flutuates
greatly between epochs, it might be that lr is too high."
```

```
"Result: 0.41 accuracy, 0.240 training loss, 0.240 validation loss"
"Discussion: This is an improvement from second try, however, it still
performed worse than lr = 1e-4"

"Fourth try: Decrease batchsize to 16, in hopes of reducing erroneous
generalization"
"Result: 0.51 accuracy, 0.237 training loss, 0.237 validation loss"
"Discussion: Significant improvement in accuracy, with highest
accuracies clustering in earlier epochs. Marginal improvement in
loss."
```

```
{"type":"string"}
```

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

```
get_accuracy(model, test_loader)
```

```
0.4767795138888889
```

Part (b) [4 pt]

Based on the test accuracy alone, it is difficult to assess whether our model is actually performing well. We don't know whether a high accuracy is due to the simplicity of the problem, or if a poor accuracy is a result of the inherent difficulty of the problem.

It is therefore very important to be able to compare our model to at least one alternative. In particular, we consider a simple **baseline** model that is not very computationally expensive. Our neural network should at least outperform this baseline model. If our network is not much better than the baseline, then it is not doing well.

For our data imputation problem, consider the following baseline model: to predict a missing feature, the baseline model will look at the **most common value** of the feature in the training set.

For example, if the feature "marriage" is missing, then this model's prediction will be the most common value for "marriage" in the training set, which happens to be "Married-civ-spouse".

What would be the test accuracy of this baseline model?

```
"If we assume that each individual can be predicted using the same
statistical probability,"
"then, the likelihood of producing a correct/accurate prediction
should be: "
"number of this most common value / total value. I.e. if the most
```

```
common value occurred 300 times,"
"out of a total of 500, then the probability of being accurate is 0.6"
```

Part (c) [1 pt]

How does your test accuracy from part (a) compared to your basline test accuracy in part (b)?

```
baseline_acc = df['edu'].value_counts().max() / len(df['edu'])
model_acc = get_accuracy(model, test_loader)

print("Baseline accuracy: ", baseline_acc)
print("Model accuracy: ", model_acc)
# the test accuracy is significantly higher than the baseline accuracy
```

```
Baseline accuracy:  0.32250238014802984
Model accuracy:  0.4767795138888889
```

Part (d) [1 pt]

Look at the first item in your test data. Do you think it is reasonable for a human to be able to guess this person's education level based on their other features? Explain.

```
get_features(test_set)
# I am not sure why the list index is out of range, when I've already
accounted for cat_index
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
last)
<ipython-input-113-91416b1aacb7> in <cell line: 1>()
----> 1 get_features(test_set)

<ipython-input-18-ec6c11b0f920> in get_features(record)
     13     Return a dictionary of all categorical feature values of a
record
     14     """
---> 15     return { f: get_feature(record, f) for f in catcols }

<ipython-input-18-ec6c11b0f920> in <dictcomp>(.0)
     13     Return a dictionary of all categorical feature values of a
record
     14     """
---> 15     return { f: get_feature(record, f) for f in catcols }

<ipython-input-18-ec6c11b0f920> in get_feature(record, feature)
      7     """
      8     onehot = get_onehot(record, feature)
----> 9     return get_categorical_value(onehot, feature)
```

```
     10
     11 def get_features(record):

<ipython-input-112-b75c0a80cc2e> in get_categorical_value(onehot,
feature)
     55       return cat_values["work"][np.argmax(onehot)]
     56     elif feature == "marriage":
---> 57       return cat_values["marriage"][np.argmax(onehot) +
cat_index["marriage"]]
     58     elif feature == "occupation":
     59       return cat_values["occupation"][np.argmax(onehot) +
cat_index["occupation"]]

IndexError: list index out of range
```

Part (e) [2 pt]

What is your model's prediction of this person's education level, given their other features?

```
model.eval()
test_df = pd.DataFrame(test_set)
first_row = test_df.iloc[0].copy()
input_tensor = torch.tensor(first_row)
with torch.no_grad():
    output = model(input_tensor)

get_features(output)

#same issue as above
```

```
---------------------------------------------------------------------------
-----
IndexError                                Traceback (most recent call
last)
<ipython-input-117-3c3164b3c461> in <cell line: 8>()
      6     output = model(input_tensor)
      7
----> 8 get_features(output)
      9
     10 #same issue as above

<ipython-input-18-ec6c11b0f920> in get_features(record)
     13     Return a dictionary of all categorical feature values of a
record
     14     """
---> 15     return { f: get_feature(record, f) for f in catcols }

<ipython-input-18-ec6c11b0f920> in <dictcomp>(.0)
     13     Return a dictionary of all categorical feature values of a
record
```

```
     14     """
---> 15     return { f: get_feature(record, f) for f in catcols }

<ipython-input-18-ec6c11b0f920> in get_feature(record, feature)
      7     """
      8     onehot = get_onehot(record, feature)
----> 9     return get_categorical_value(onehot, feature)
     10
     11 def get_features(record):

<ipython-input-112-b75c0a80cc2e> in get_categorical_value(onehot,
feature)
     55       return cat_values["work"][np.argmax(onehot)]
     56     elif feature == "marriage":
---> 57       return cat_values["marriage"][np.argmax(onehot) +
cat_index["marriage"]]
     58     elif feature == "occupation":
     59       return cat_values["occupation"][np.argmax(onehot) +
cat_index["occupation"]]

IndexError: list index out of range
```

Part (f) [2 pt]

What is the baseline model's prediction of this person's education level?

```
"the baseline prediction would be HS-grad, since it is the most common
value in the dataset given"
```


```
{"type":"string"}
```