Titanic Supervised Learning

We need two key data files that you will work with - a training set and a testing set.

In [1]:

```
import pandas as pd

test = pd.read_csv("../input/test.csv")
test_shape = test.shape
print(test_shape)
```

```
(418, 11)
```

Instructions

Use `pandas.read_csv()` to import `train.csv` and assign it to the variable `train`.
Use `DataFrame.shape` to calculate the number of rows and columns in `train`, and assign the result to `train_shape`.

In [2]:

```
# INSERT CODE HERE
train = pd.read_csv("../input/train.csv")
train_shape = train.shape
print(train_shape)
```

```
(891, 12)
```

2. Exploring the data

Below are the descriptions contained in that data dictionary:

- *PassengerID* - A column to identify each row and make submissions easier
- *Survived* - Whether the passenger survived or not and the value we are predicting (0=No, 1=Yes)
- *Pclass* - The class of the ticket the passenger purchased (1=1st, 2=2nd, 3=3rd)
- *Sex* - The passenger's sex
- *Age* - The passenger's age in years
- *SibSp* - The number of siblings or spouses the passenger had aboard the Titanic
- *Parch* - The number of parents or children the passenger had aboard the Titanic
- *Ticket* - The passenger's ticket number
- *Fare* - The fare the passenger paid
- *Cabin* - The passenger's cabin number
- *Embarked* - The port where the passenger embarked (C=Cherbourg, Q=Queenstown, S=Southampton)

Let's get a view of the actual data

In [3]:

```
train.head(10)
```

Out[3]:

The type of machine learning we will be doing is called classification, because when we make predictions we are classifying each passenger as survived or not. More specifically,

we are performing binary classification, which means that there are only two different states we are classifying.

In any machine learning exercise, thinking about the topic you are predicting is very important. We call this step acquiring domain knowledge, and it's one of the most important determinants for success in machine learning.

In this case, understanding the Titanic disaster and specifically what variables might affect the outcome of survival is important. Anyone who has watched the movie Titanic would remember that women and children were given preference to lifeboats (as they were in real life). You would also remember the vast class disparity of the passengers.

This indicates that Age, Sex, and PClass may be good predictors of survival. We'll start by exploring Sex and Pclass by visualizing the data.

Because the Survived column contains 0 if the passenger did not survive and 1 if they did, we can segment our data by sex and calculate the mean of this column. We can use DataFrame.pivot_table() to easily do this:

In [4]:

```
import matplotlib.pyplot as plt

sex_pivot = train.pivot_table(index="Sex",values="Survived")
sex_pivot

# sex_pivot.plot.bar()
# plt.show()
```

Out[4]:

	Survived
Sex	
female	0.742038
male	0.188908

We can immediately see that females survived in much higher proportions than males did.

Let's do the same with the Pclass column.

Instructions

- Use `DataFrame.pivot_table()` to pivot the `train` dataframe:
  - Use "Pclass" for the index parameter.
  - Use "Survived" for the values parameter.
- Use `DataFrame.plot.bar()` to plot the pivot table.

In [5]:

```
pclass_pivot = train.pivot_table(index="Pclass",values="Survived")
```

```
# pclass_pivot
pclass_pivot.plot.bar()
plt.show()
```

3. Exploring and Converting the age column

The Sex and PClass columns are what we call categorical features. That means that the values represented a few separate options (for instance, whether the passenger was male or female).

Let's take a look at the Age column using `Series.describe()`.

In [6]:

```
train['Age'].describe()
```

Out[6]:

```
count    714.000000
mean      29.699118
std       14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
Name: Age, dtype: float64
```

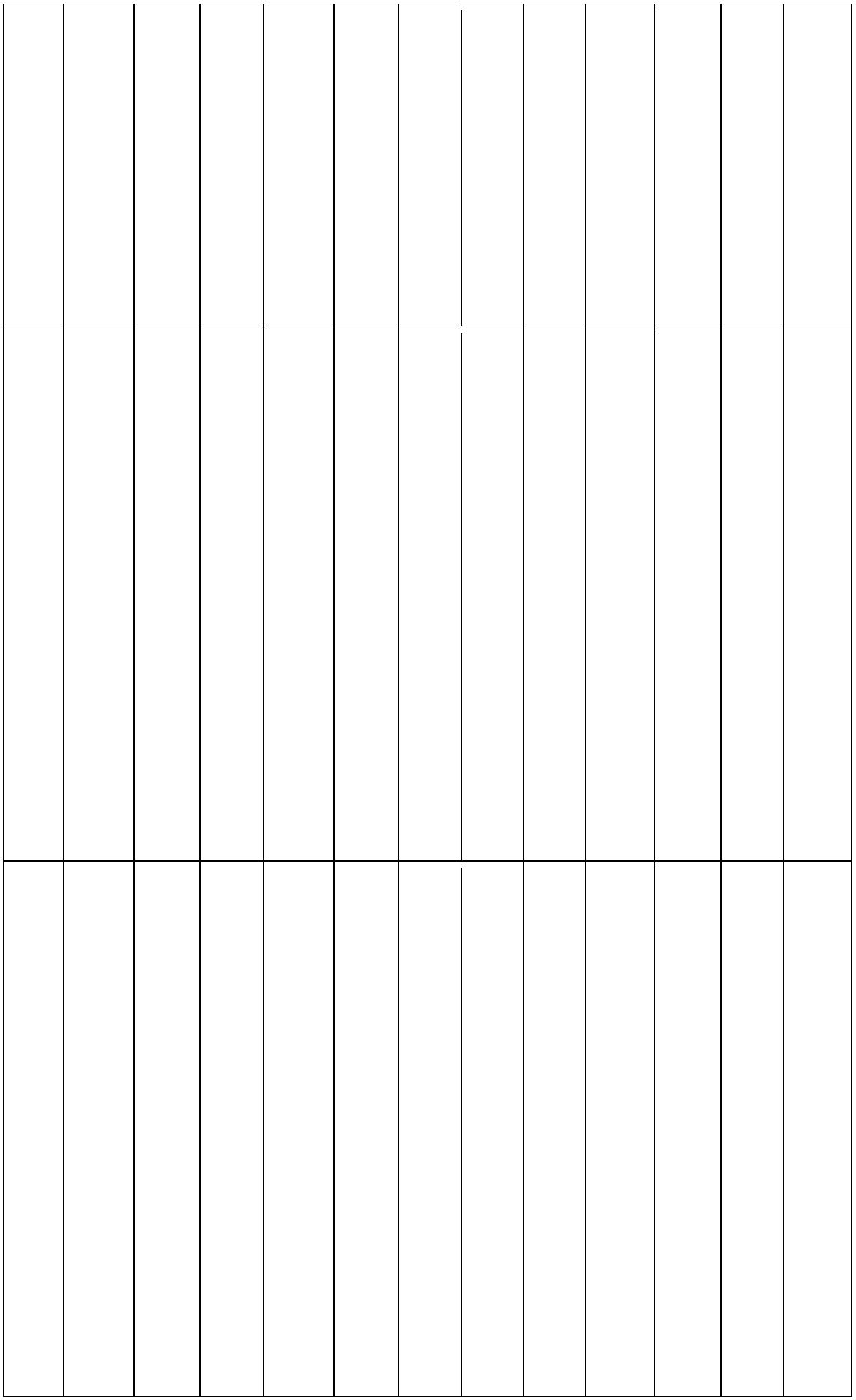
The Age column contains numbers ranging from 0.42 to 80. The other thing to note here is that there are 714 values in this column, fewer than the 814 rows we discovered that the train data set had earlier in this mission which indicates we have some missing values.

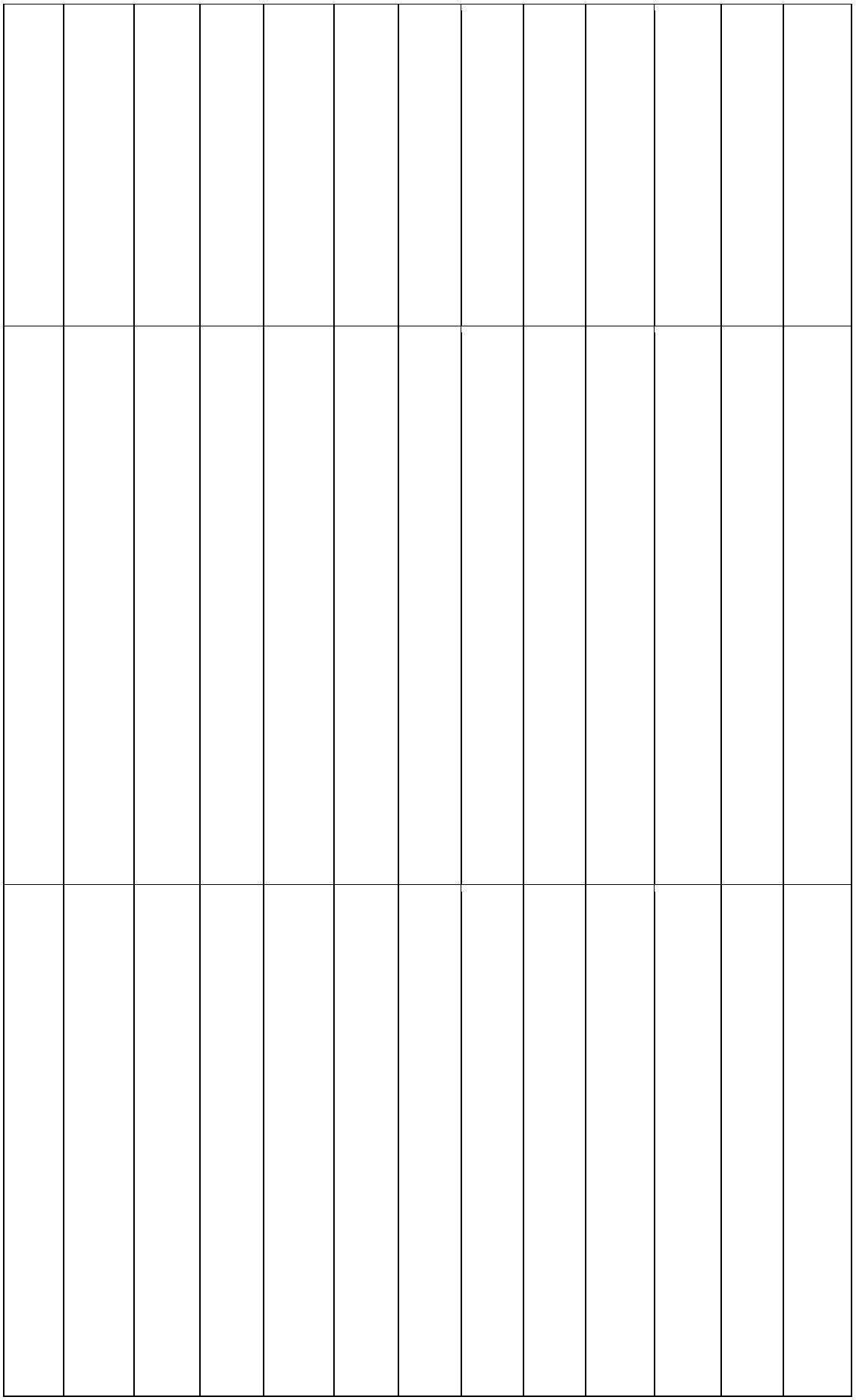
All of this means that the Age column needs to be treated slightly differently, as this is a continuous numerical column. One way to look at distribution of values in a continuous numerical set is to use histograms. We can create two histograms to compare visually the those that survived vs those who died across different age ranges:

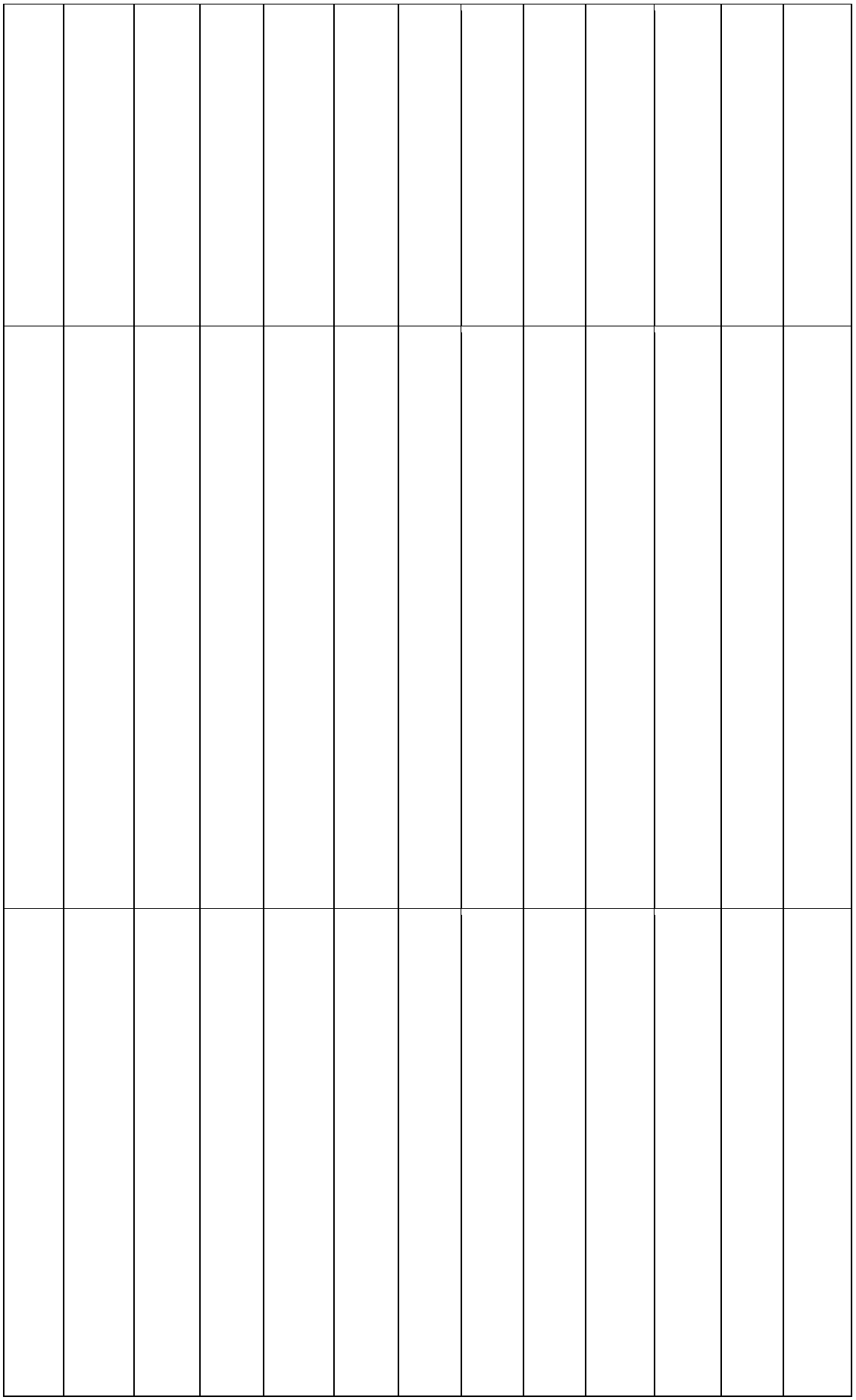
In [7]:

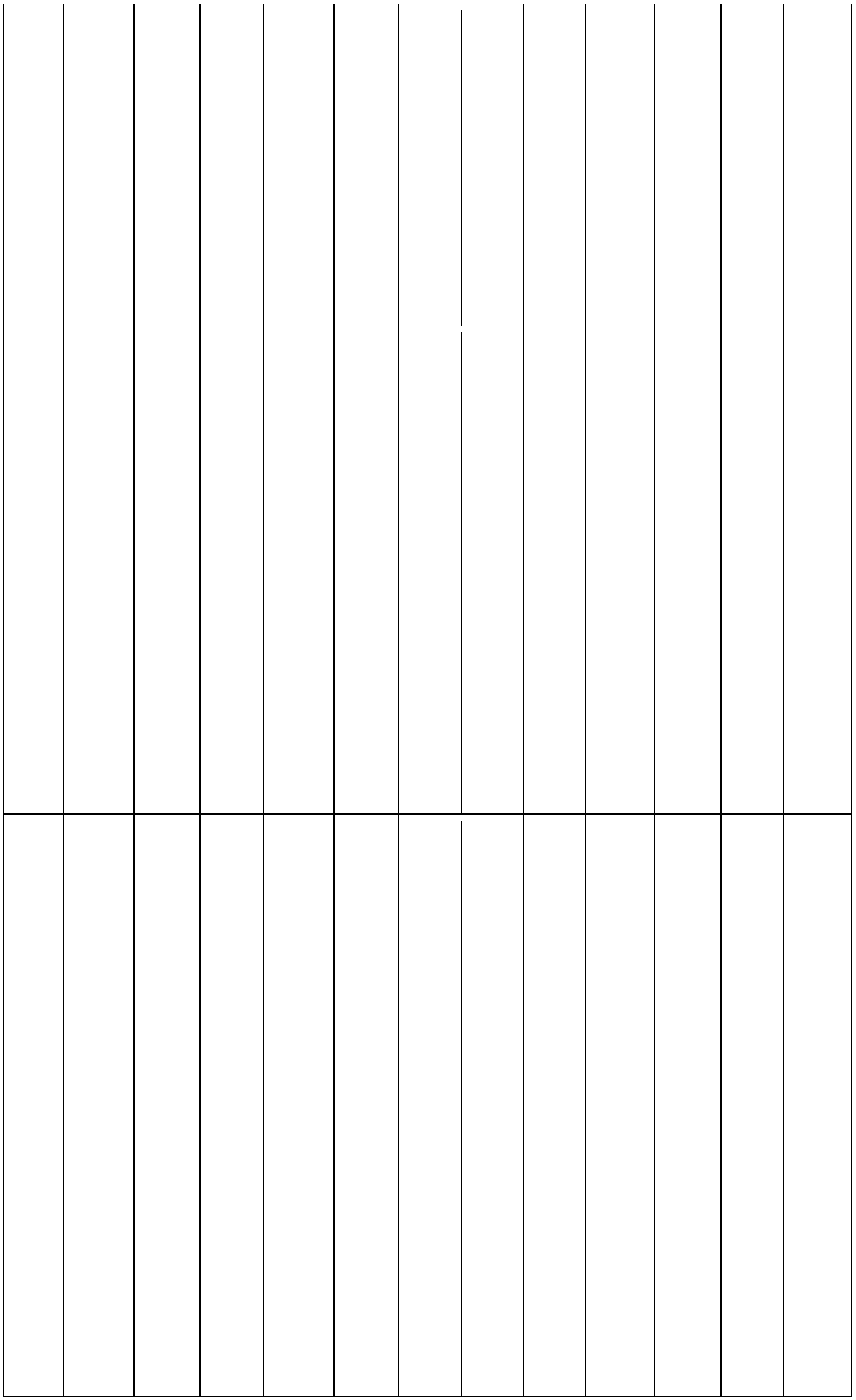
```
train[train["Survived"] == 1]
```

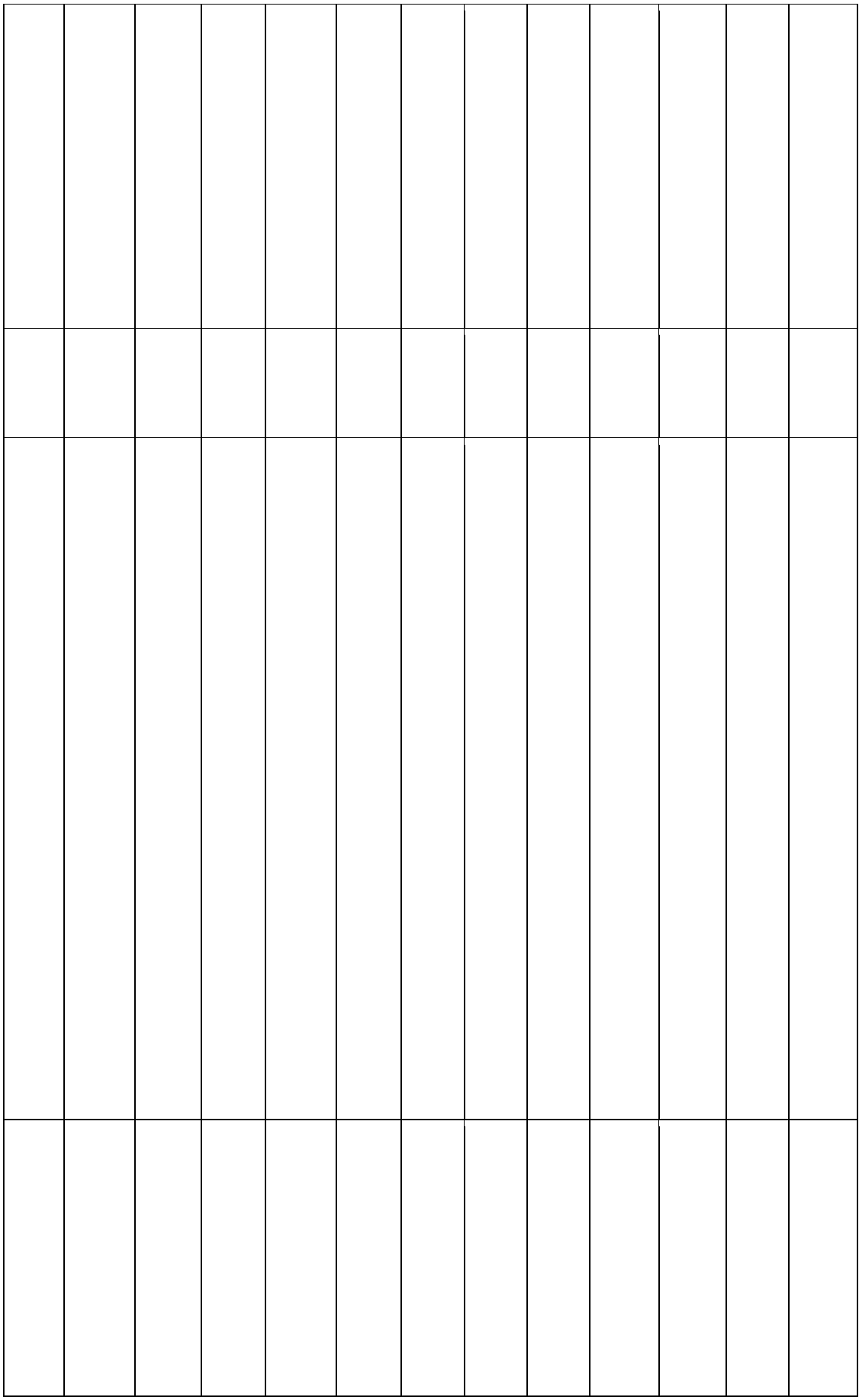
Out[7]:

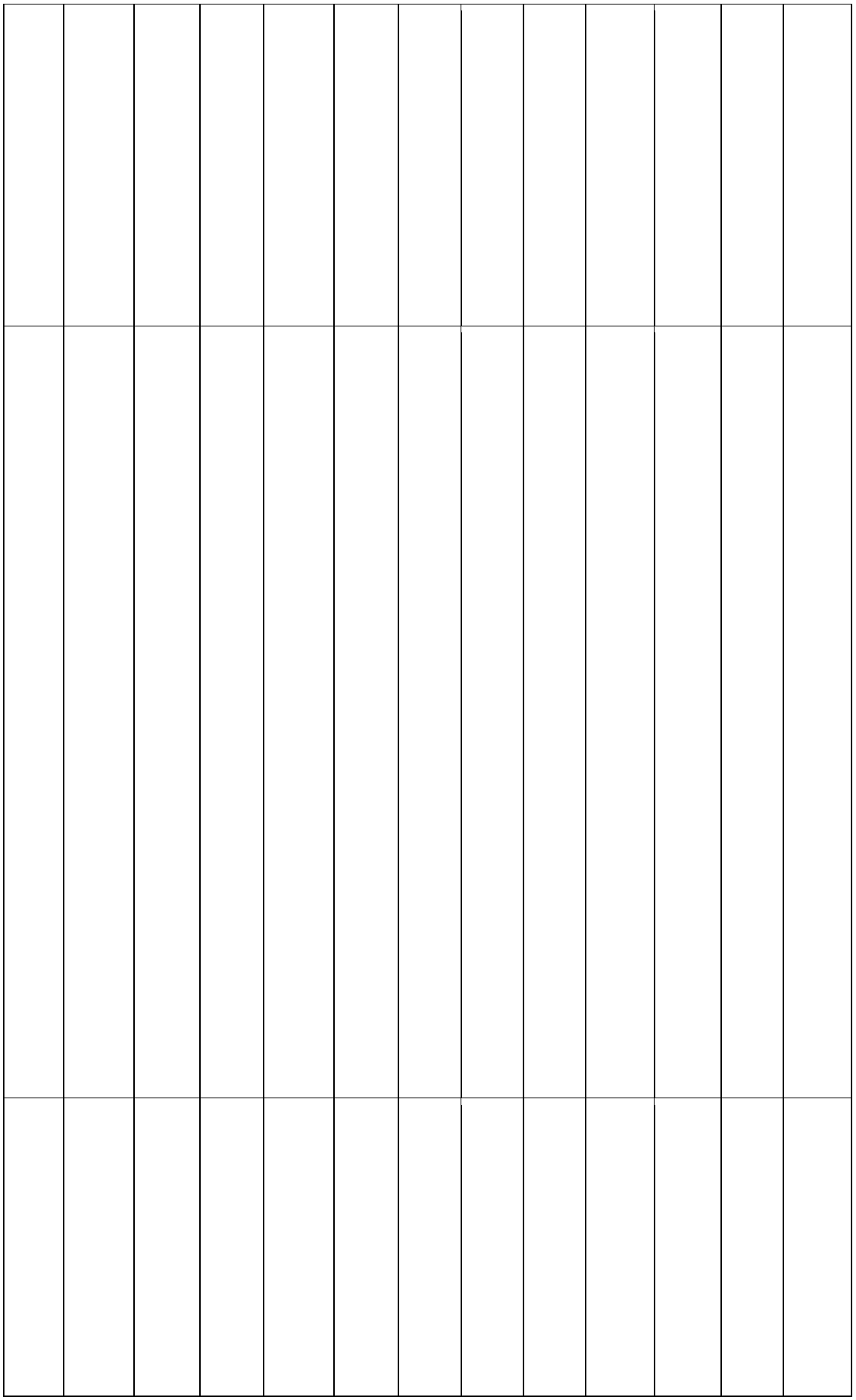


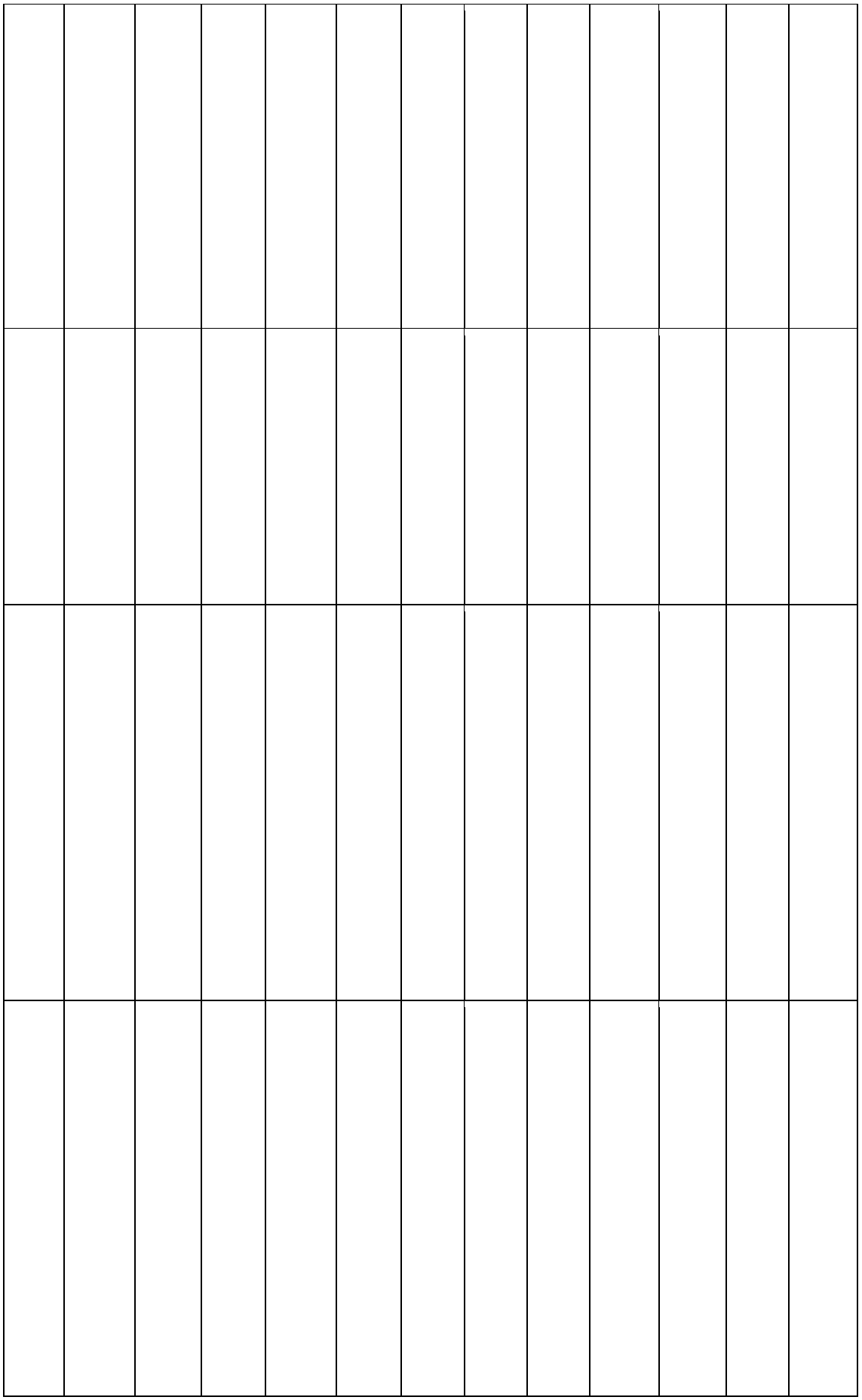


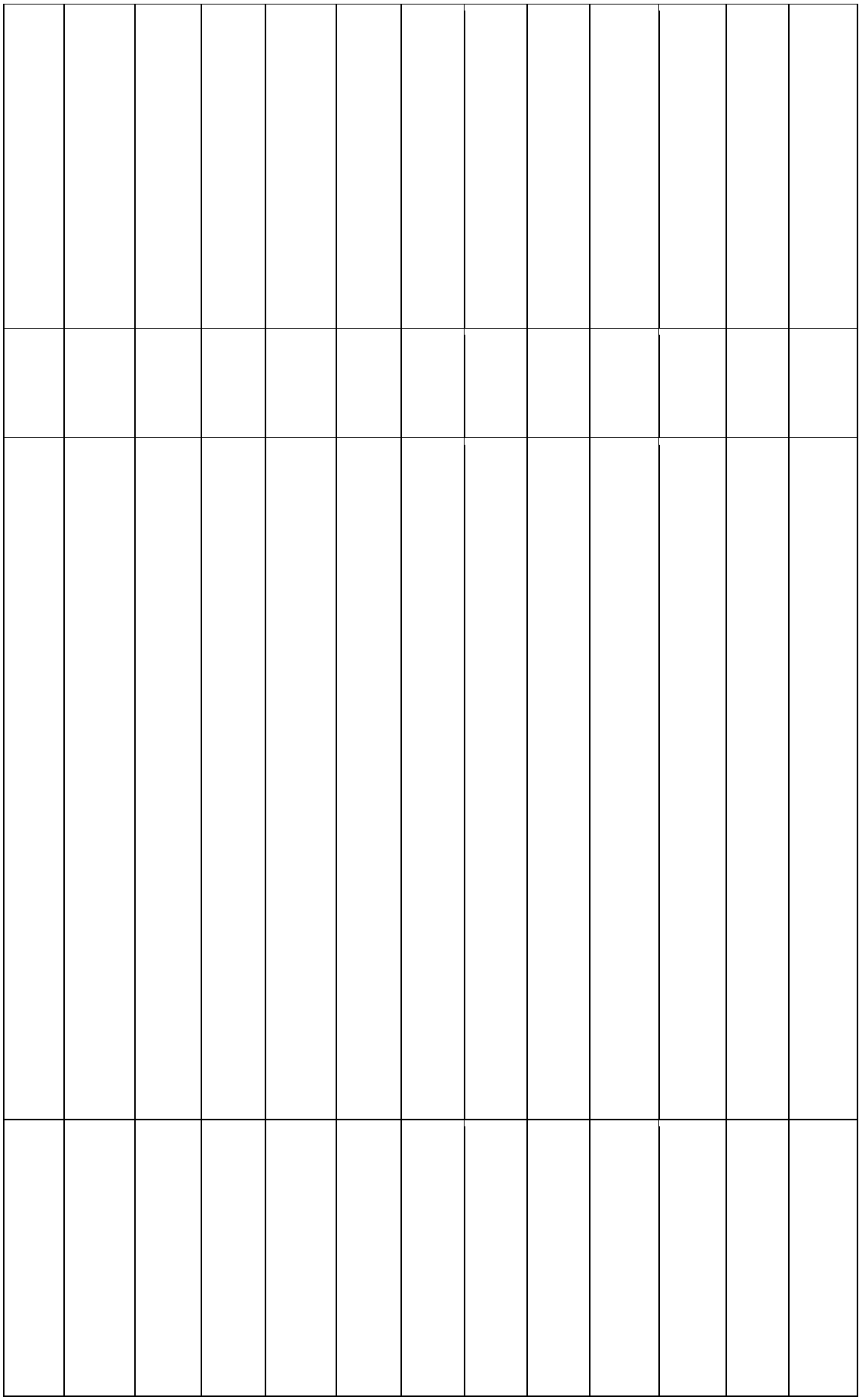


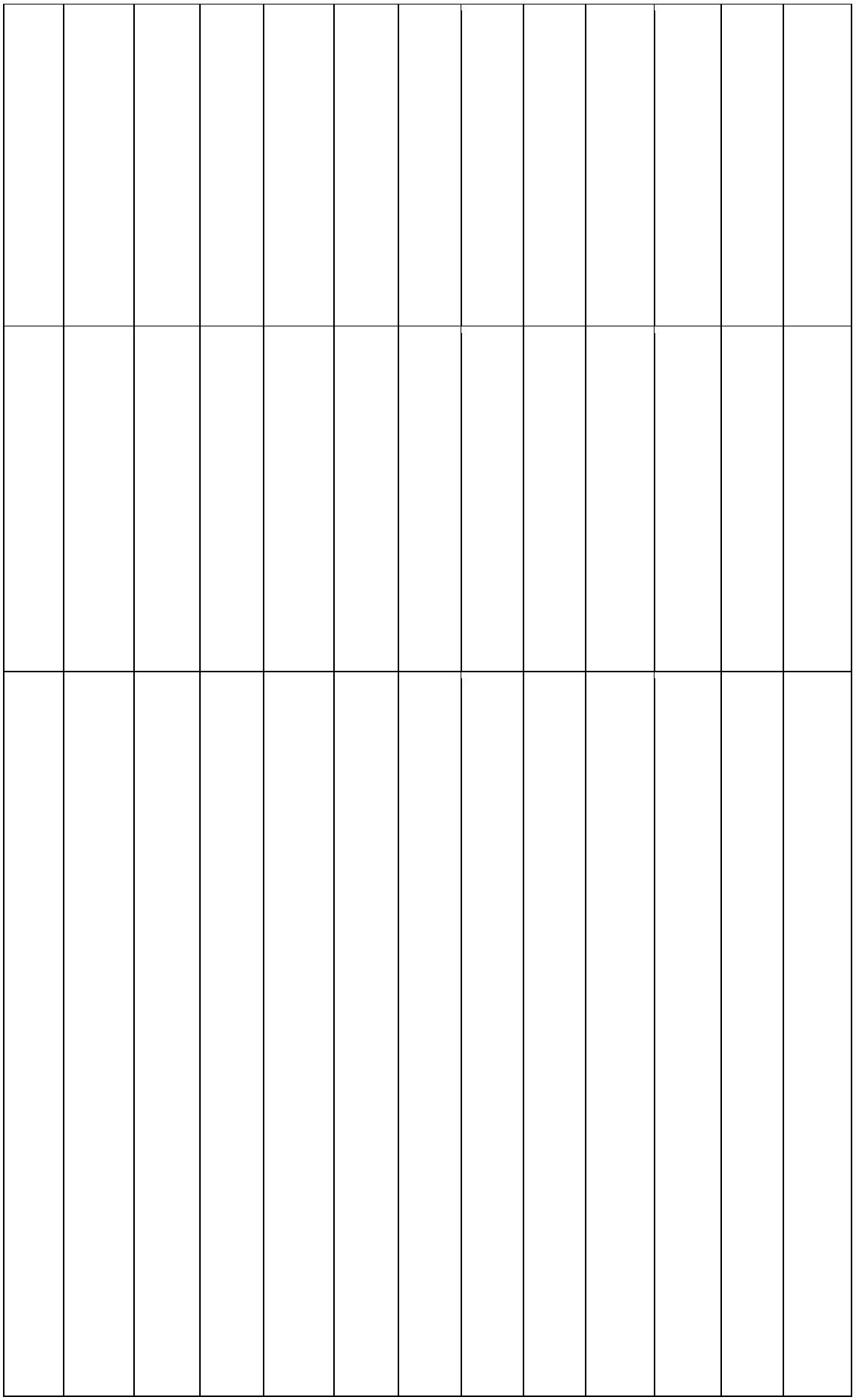


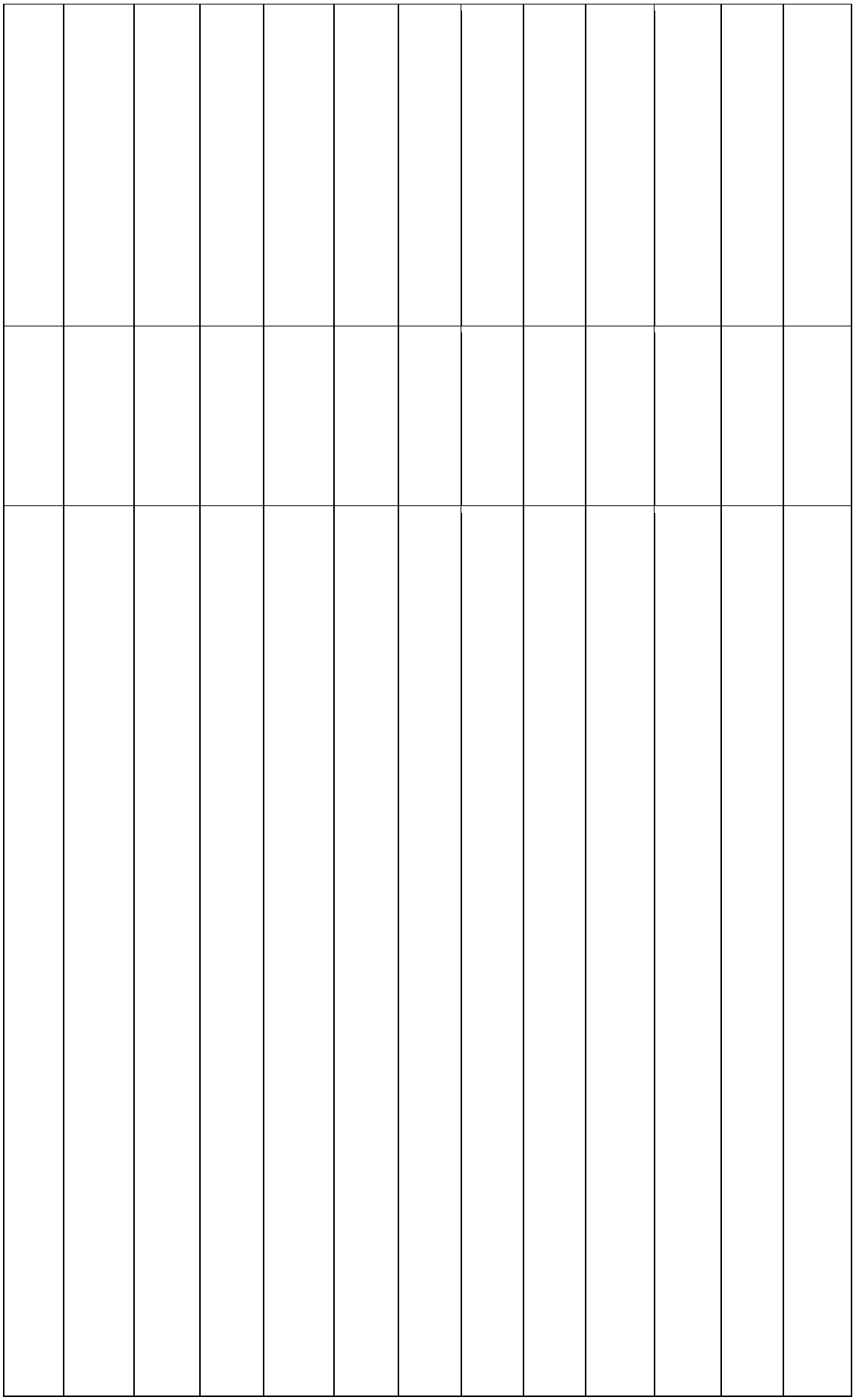


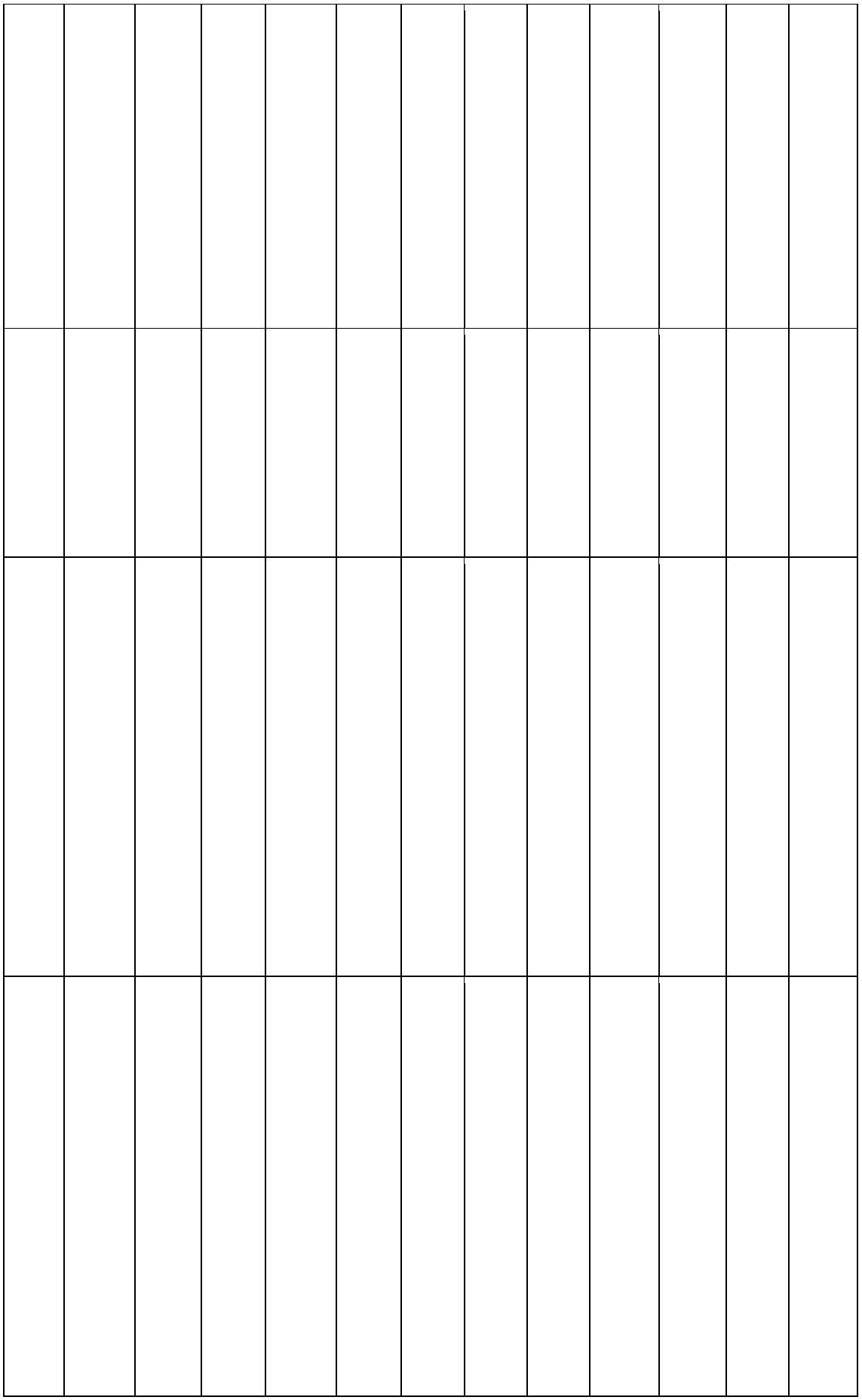


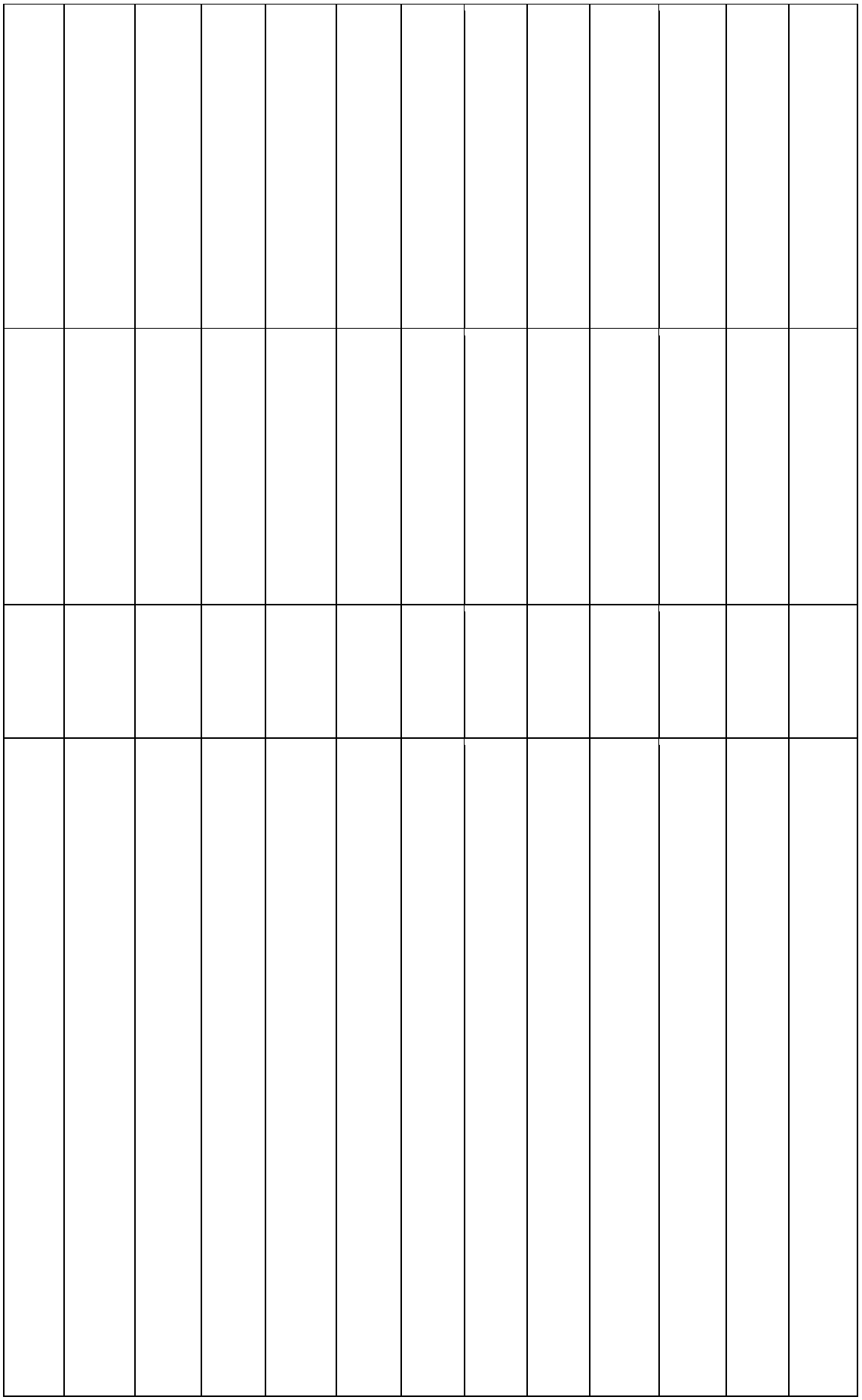


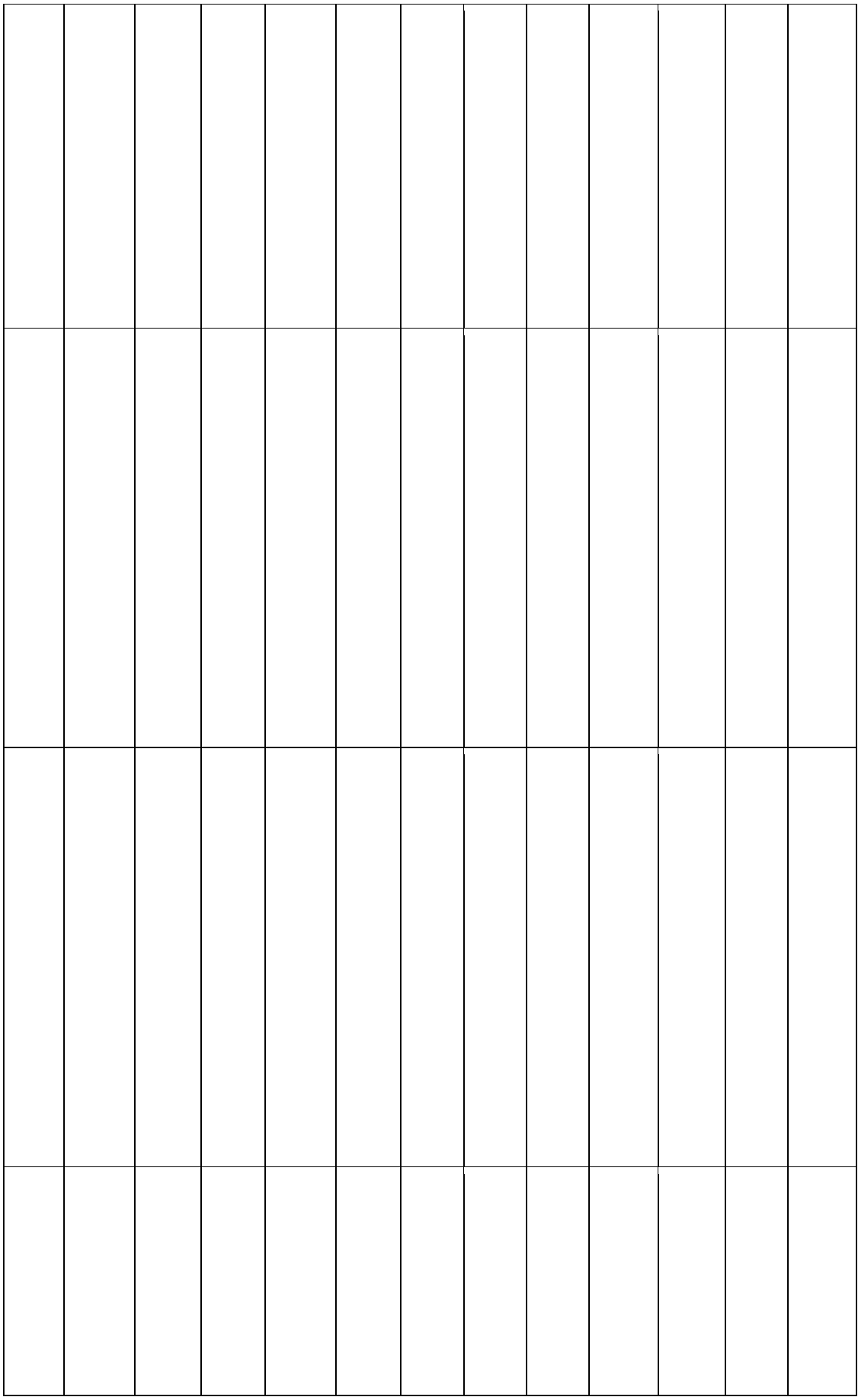


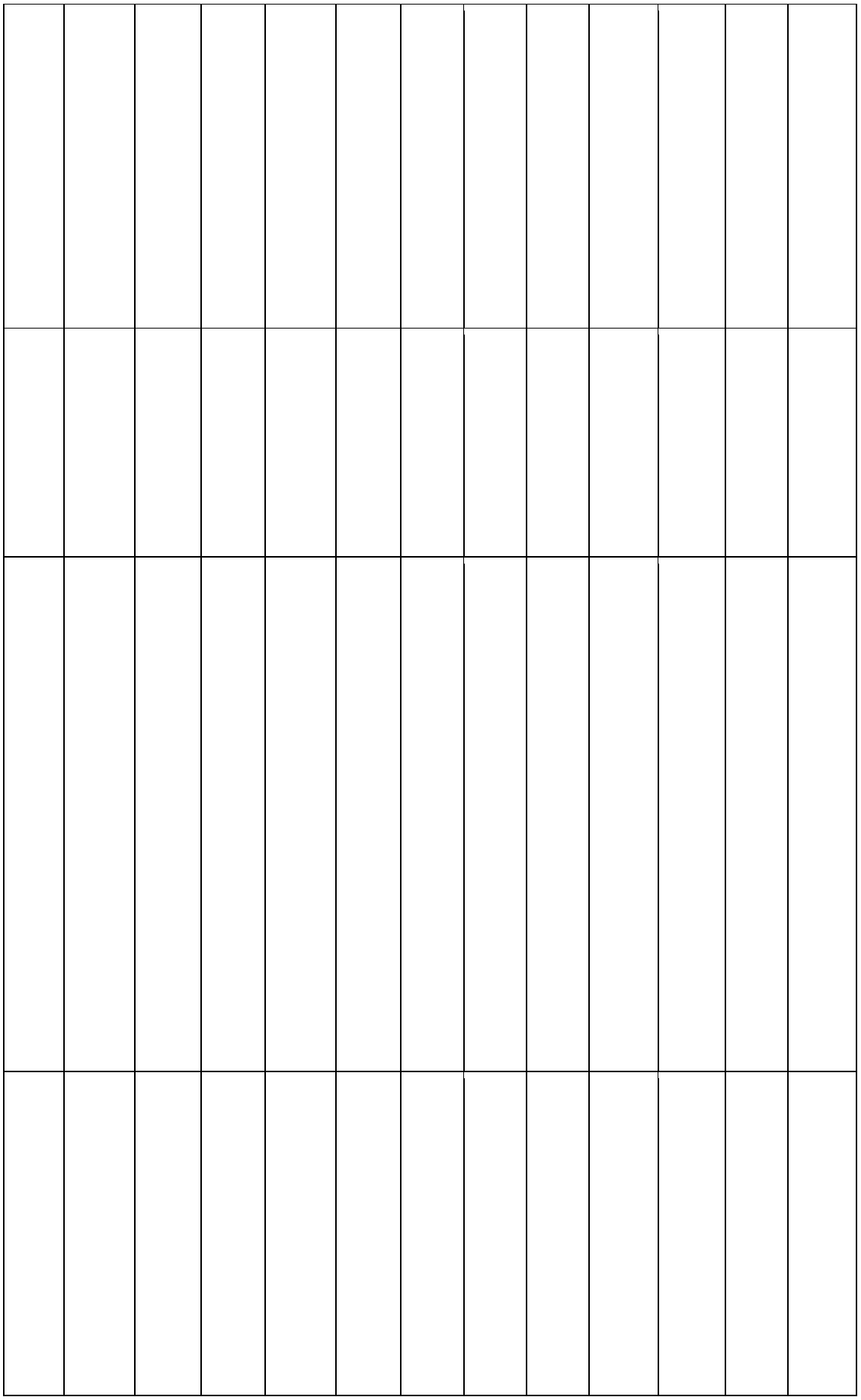


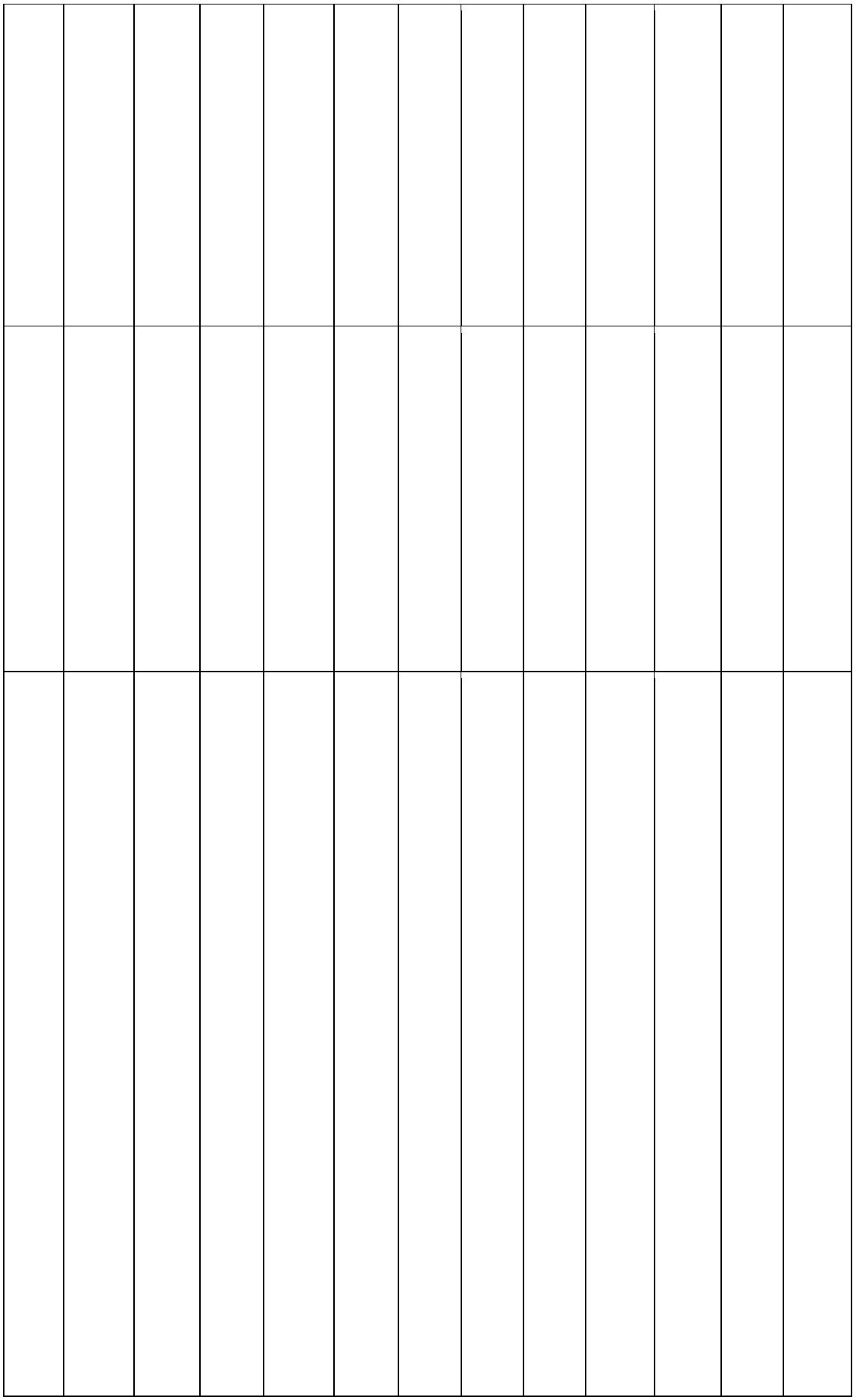


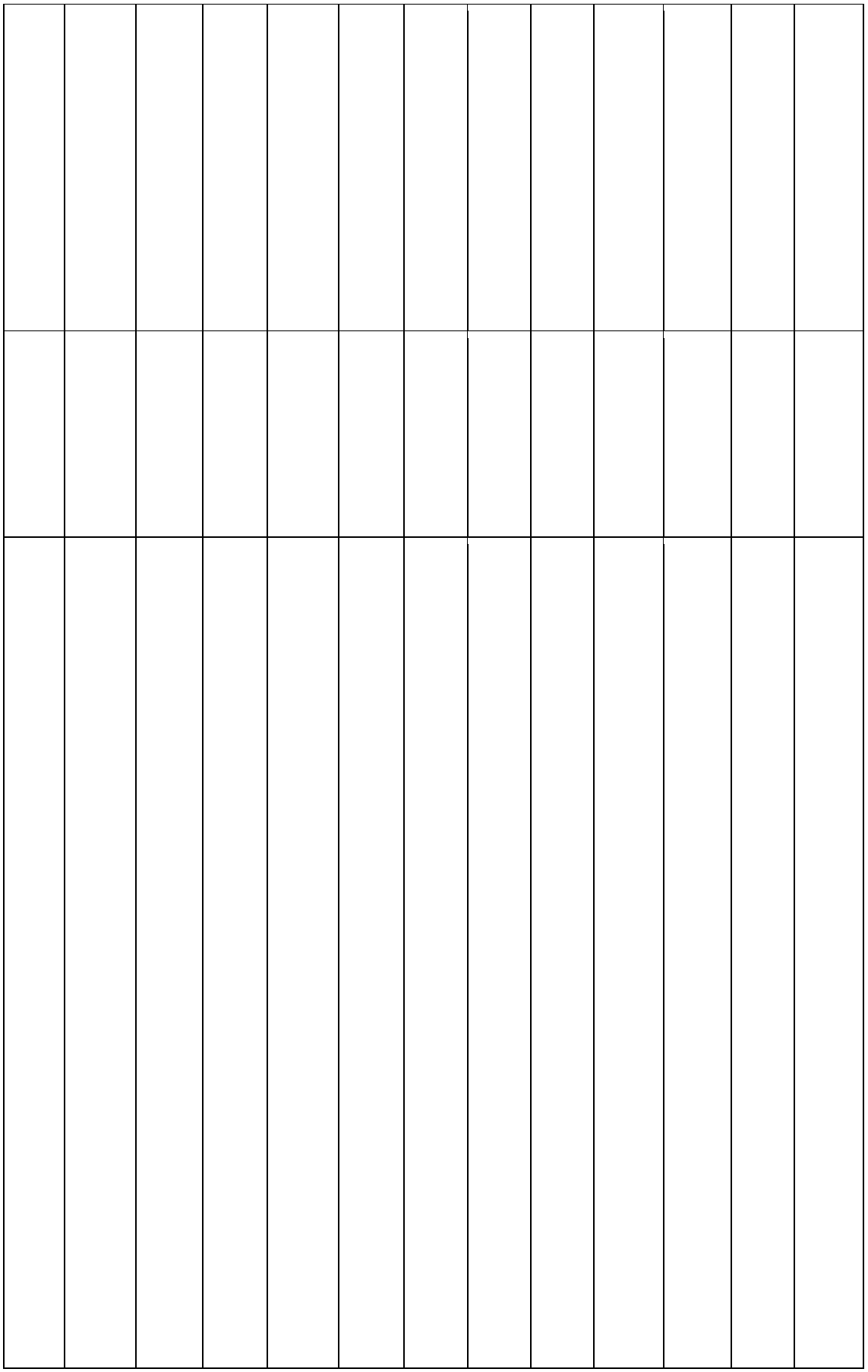


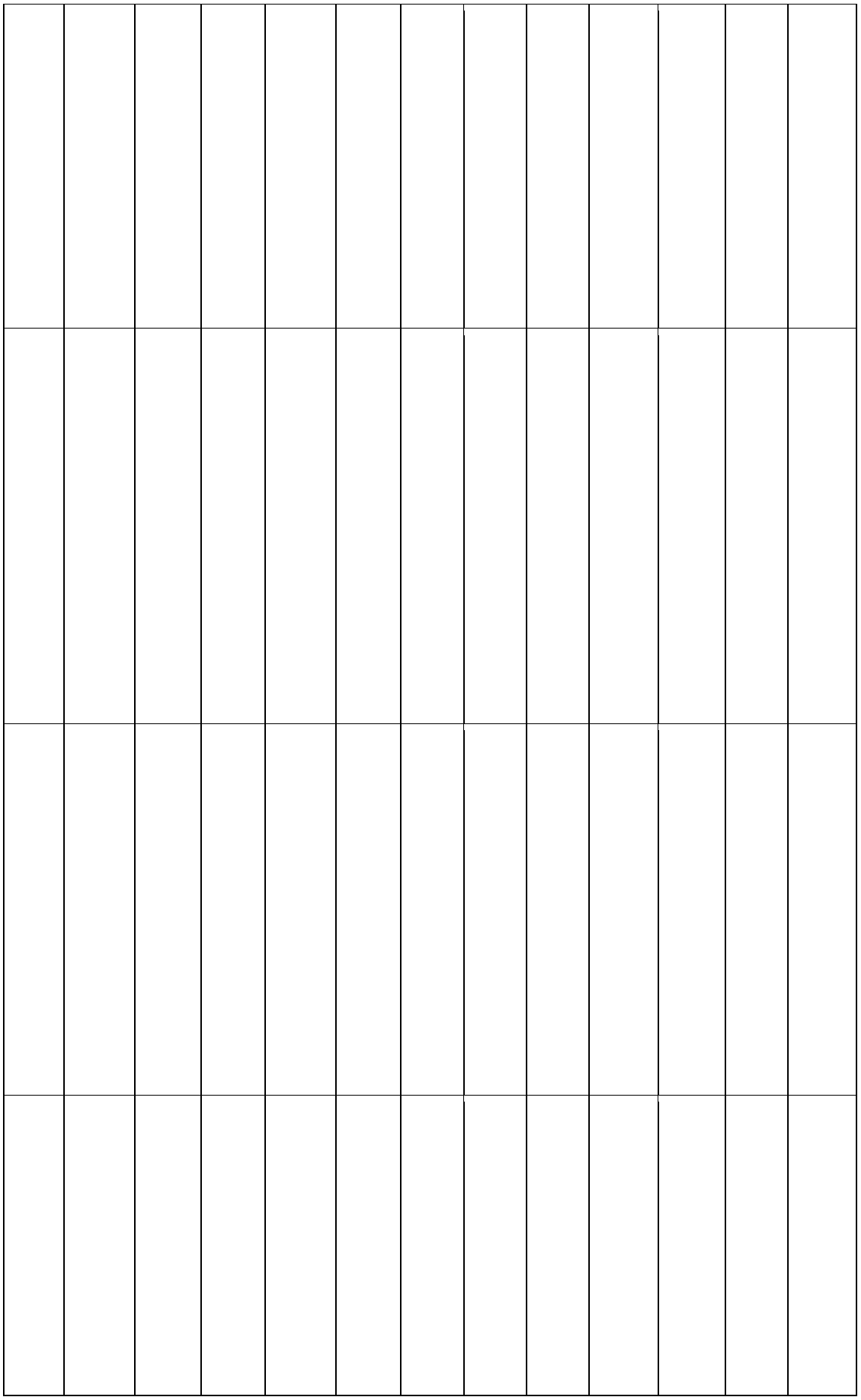


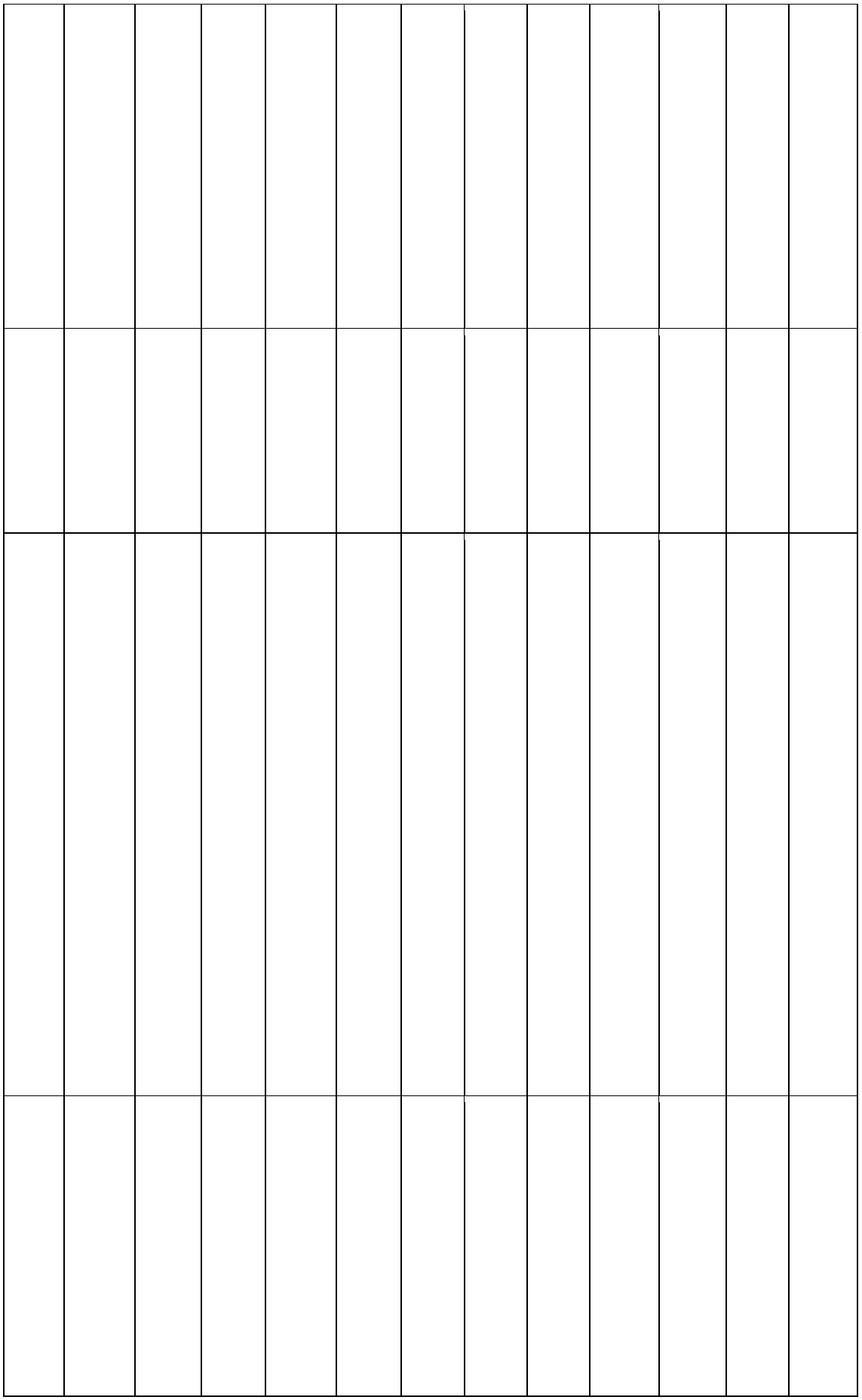


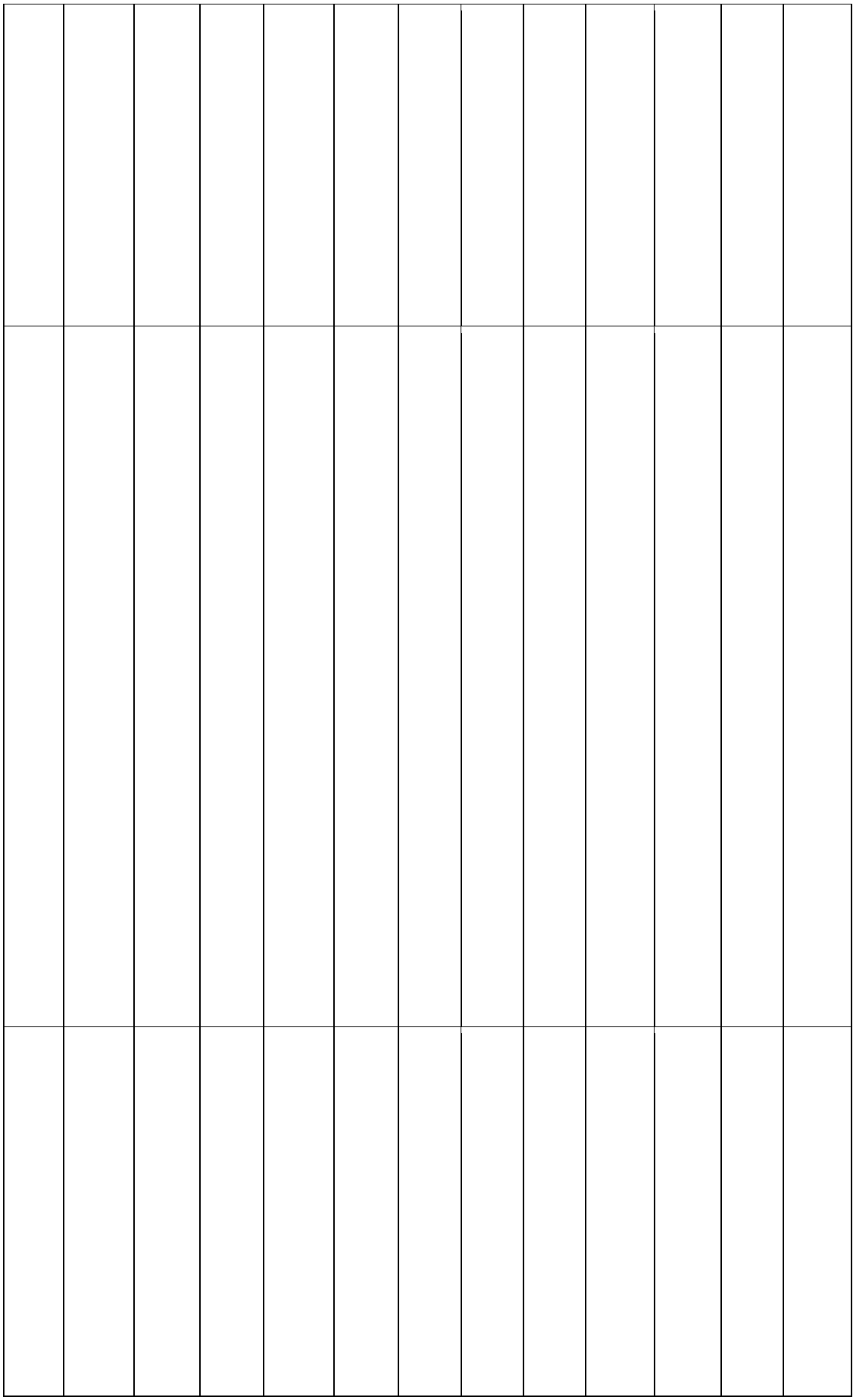


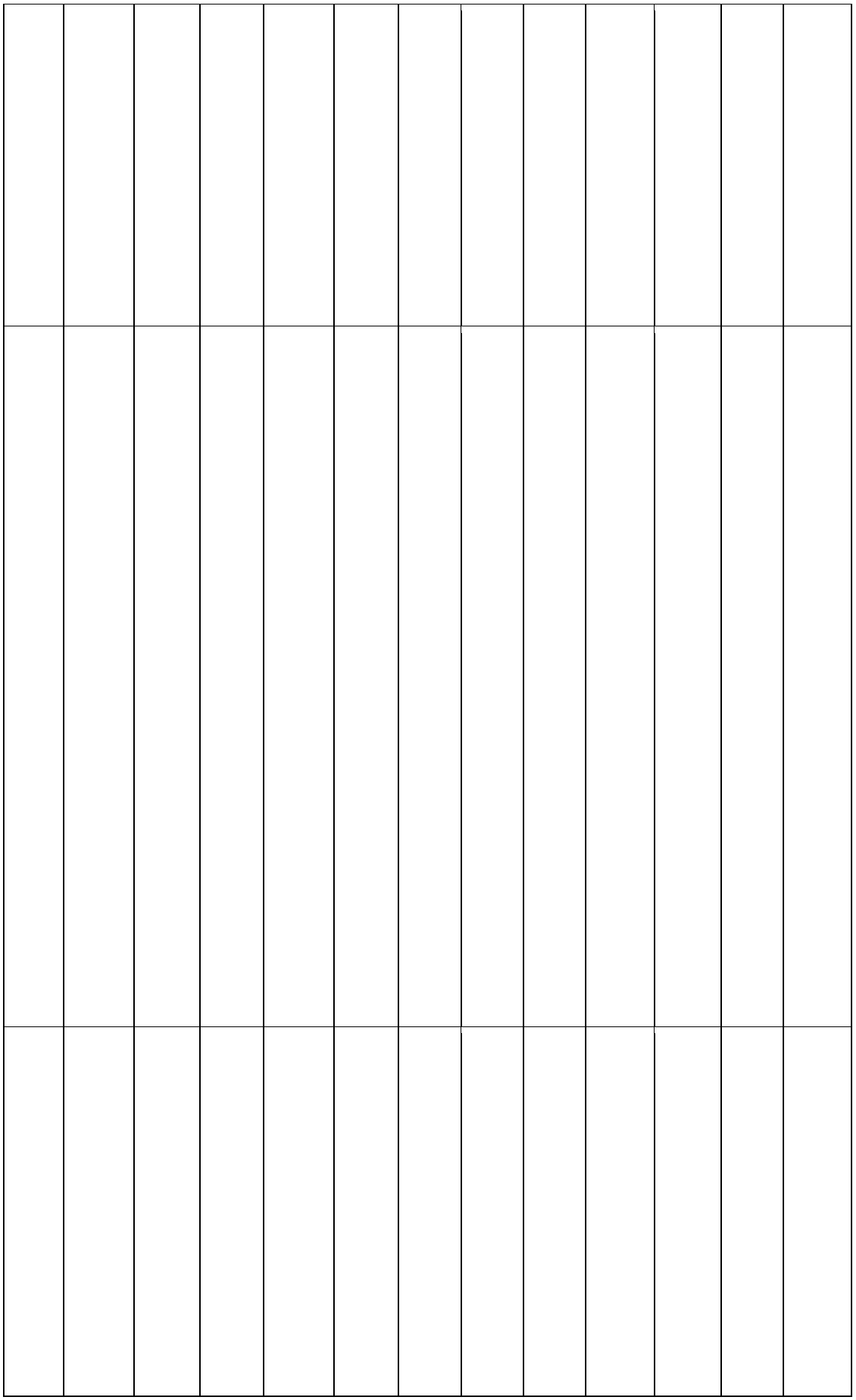


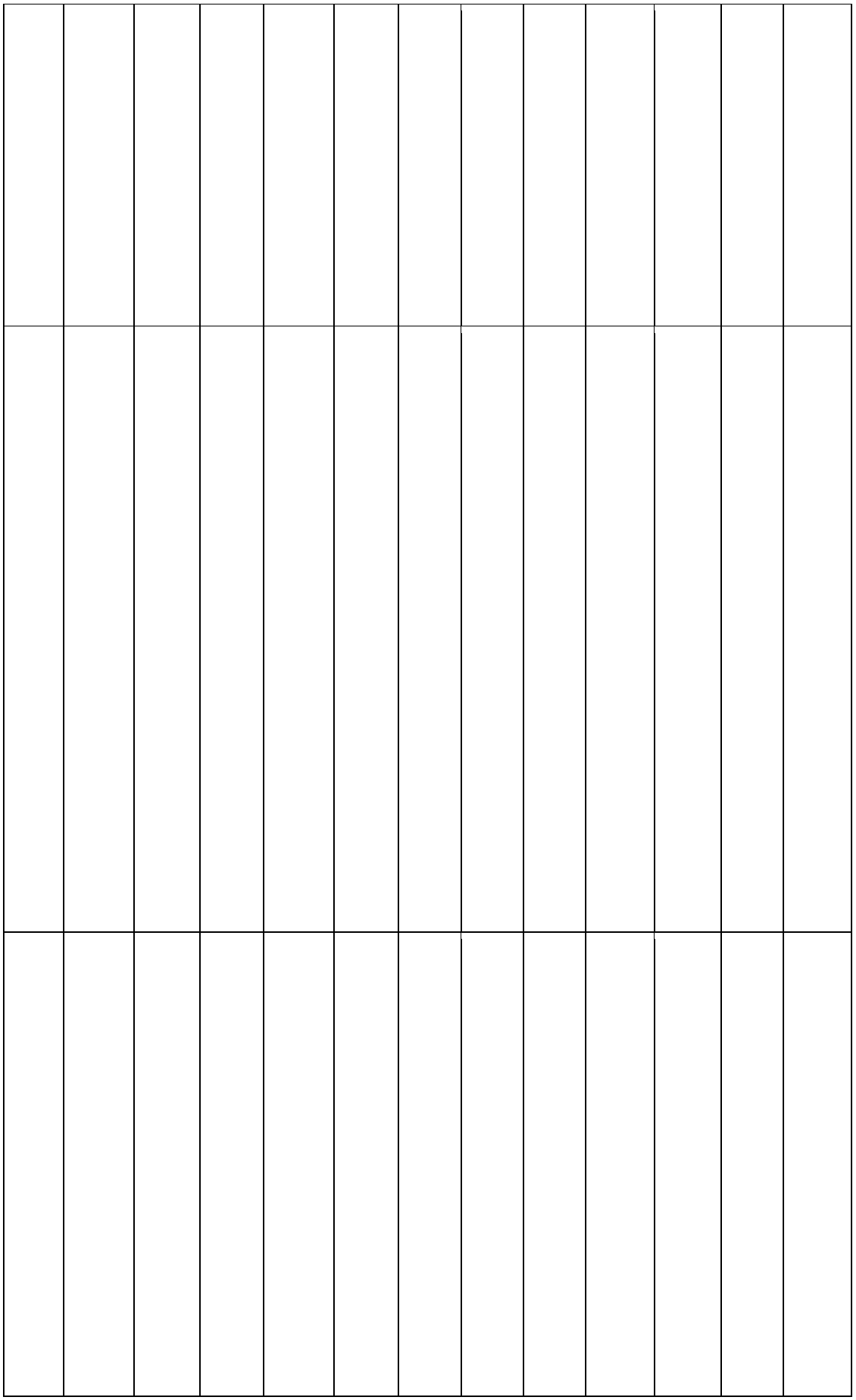


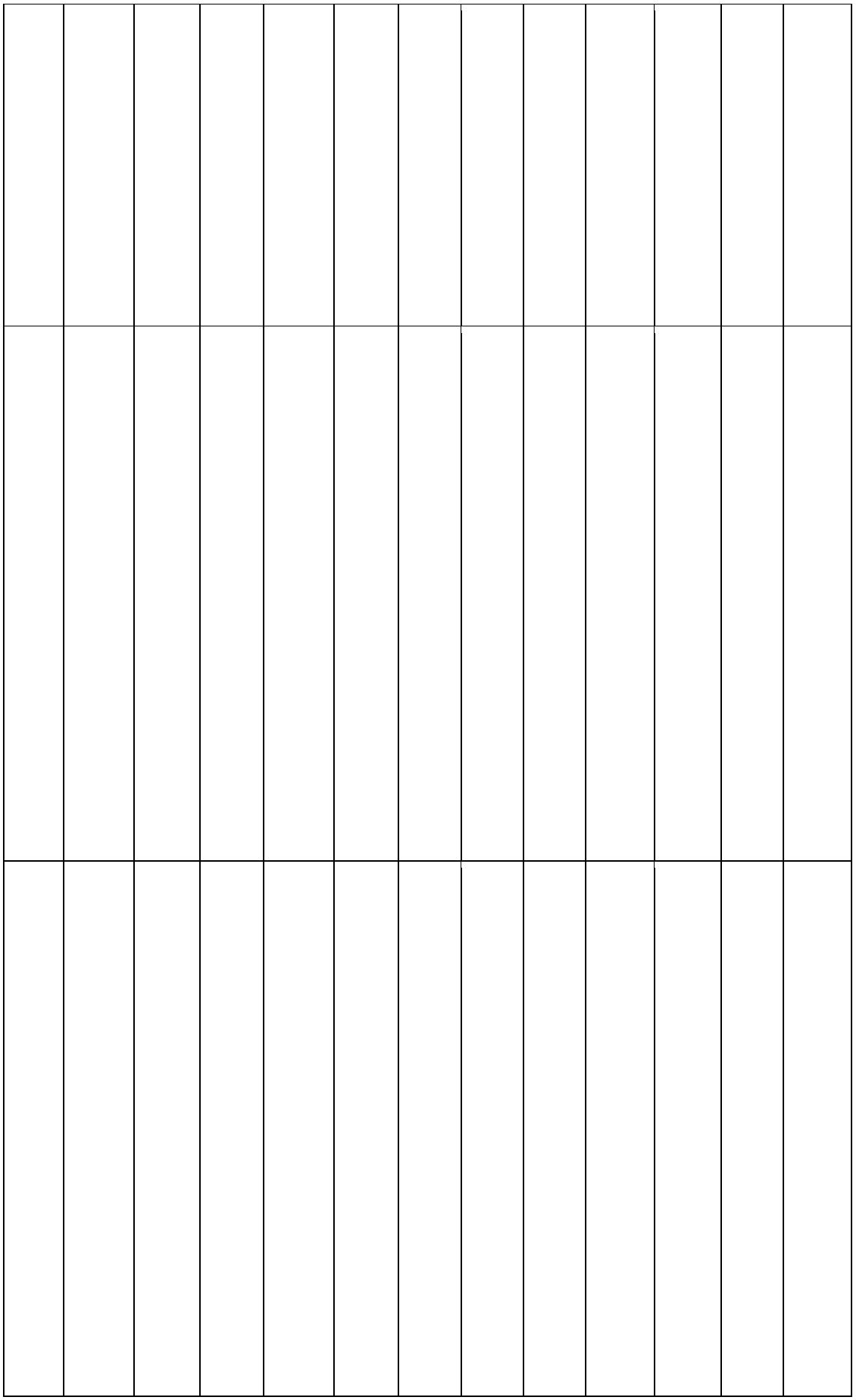












342 rows × 12 columns

In [8]:

```
survived = train[train["Survived"] == 1]
died = train[train["Survived"] == 0]
survived["Age"].plot.hist(alpha=0.5,color='red',bins=50)
died["Age"].plot.hist(alpha=0.5,color='blue',bins=50)
plt.legend(['Survived','Died'])
plt.show()
```

The relationship here is not simple, but we can see that in some age ranges more passengers survived - where the red bars are higher than the blue bars.

In order for this to be useful to our machine learning model, we can separate this continuous feature into a categorical feature by dividing it into ranges. We can use the `pandas.cut()` function to help us out.

The `pandas.cut()` function has two required parameters - the column we wish to cut, and a list of numbers which define the boundaries of our cuts. We are also going to use the

optional parameter labels, which takes a list of labels for the resultant bins. This will make it easier for us to understand our results.

Before we modify this column, we have to be aware of two things. Firstly, any change we make to the train data, we also need to make to the test data, otherwise we will be unable to use our model to make predictions for our submissions. Secondly, we need to remember to handle the missing values we observed above.

We can then use that function on both the `train` and `test` dataframes.

```
def process_age(df,cut_points,label_names):

    df["Age"] = df["Age"].fillna(-0.5)

    df["Age_categories"] = pd.cut(df["Age"],cut_points,labels=label_names
)

    return df



cut_points = [-1,0,18,100]

label_names = ["Missing","Child","Adult"]



train = process_age(train,cut_points,label_names)

test = process_age(test,cut_points,label_names)
```

The diagram below shows how the function converts the data:

Note that the cut_points list has one more element than the label_names list, since it needs to define the upper boundary for the last segment.

Instructions

- Create the `cut_points` and `label_names` lists to split the `Age` column into six categories:
    - Missing, from -1 to 0
    - Infant, from 0 to 5
    - Child, from 5 to 12
    - Teenager, from 12 to 18
    - Young Adult, from 18 to 35
    - Adult, from 35 to 60
    - Senior, from 60 to 100
- Apply the `process_age()` function on the train dataframe, assigning the result to train.
- Apply the `process_age()` function on the test dataframe, assigning the result to test.
- Use `DataFrame.pivot_table()` to pivot the train dataframe by the Age_categories column.

- Use `DataFrame.plot.bar()` to plot the pivot table.

```
In [9]:
def process_age(df,cut_points,label_names):
    df["Age"] = df["Age"].fillna(-0.5)
    df["Age_categories"] = pd.cut(df["Age"],cut_points,labels=label_names)
    return df

cut_points = [-1,0, 5, 12, 18, 35, 60, 100]
label_names = ["Missing", 'Infant', "Child", 'Teenager', "Young Adult", 'Adul
t', 'Senior']

train = process_age(train,cut_points,label_names)
test = process_age(test,cut_points,label_names)

age_cat_pivot = train.pivot_table(index="Age_categories",values="Survived")
age_cat_pivot.plot.bar()
plt.show()
```

4. Preparing our Data for Machine Learning

So far we have identified three columns that may be useful for predicting survival:

- Sex
- Pclass
- Age, or more specifically our newly created Age_categories

Before we build our model, we need to prepare these columns for machine learning. Most machine learning algorithms can't understand text labels, so we have to convert our values into numbers.

Additionally, we need to be careful that we don't imply any numeric relationship where there isn't one. If we think of the values in the Pclass column, we know they are 1, 2, and 3.

```
In [10]:

train['Pclass'].value_counts()
Out[10]:

3    491
1    216
2    184
Name: Pclass, dtype: int64
```

While the class of each passenger certainly has some sort of ordered relationship, the relationship between each class is not the same as the relationship between the numbers 1, 2, and 3. For instance, class 2 isn't "worth" double what class 1 is, and class 3 isn't "worth" triple what class 1 is.

In order to remove this relationship, we can create dummy columns for each unique value in Pclass:

Let's use that function to create dummy columns for both the Sex and Age_categories columns.

Instructions

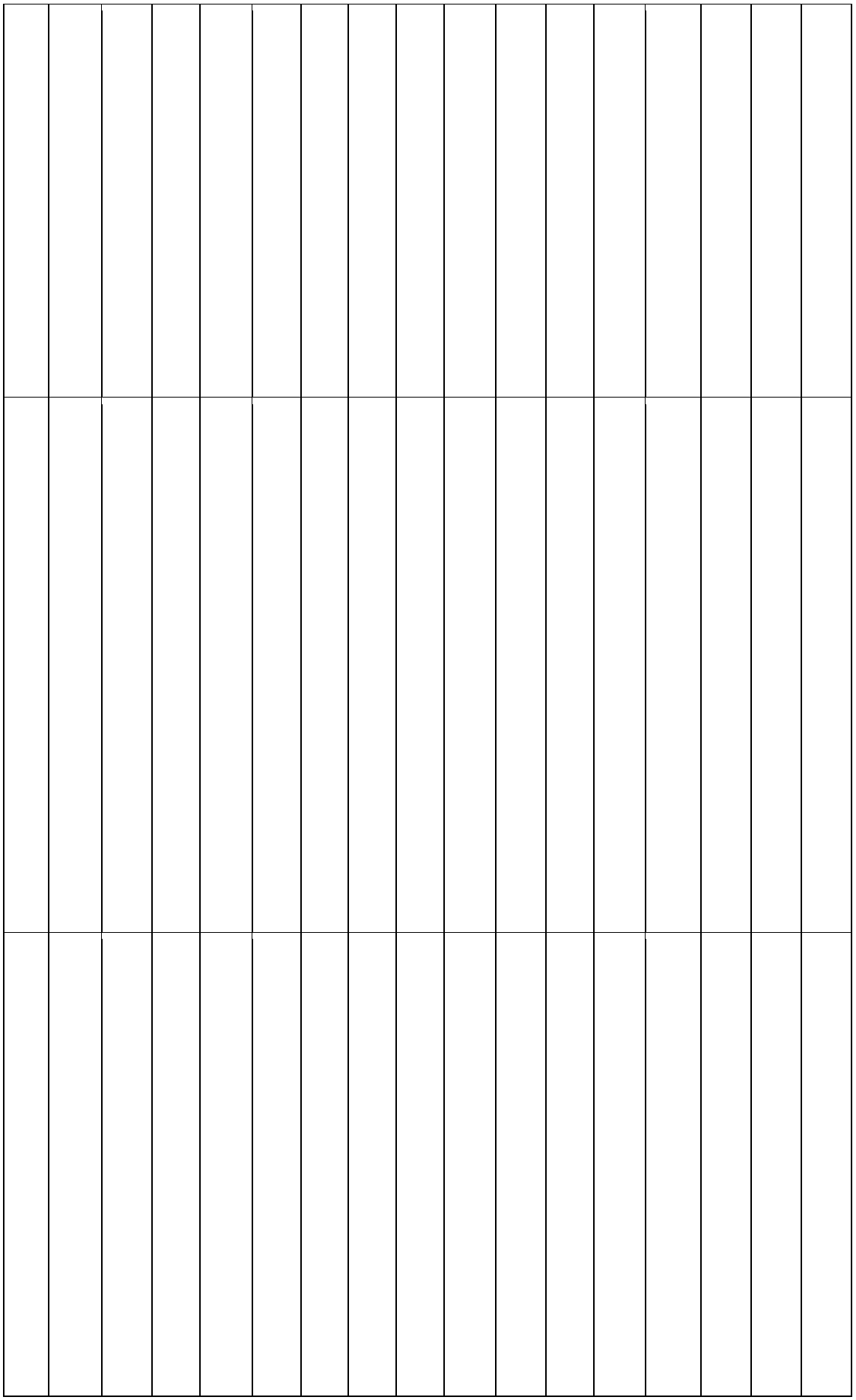
- Use the create_dummies() function to create dummy variables for the Sex column:
    - in the train dataframe.
    - in the test dataframe.
- Use the create_dummies() function to create dummy variables for the Age_categories column:
    - in the train dataframe.
    - in the test dataframe.

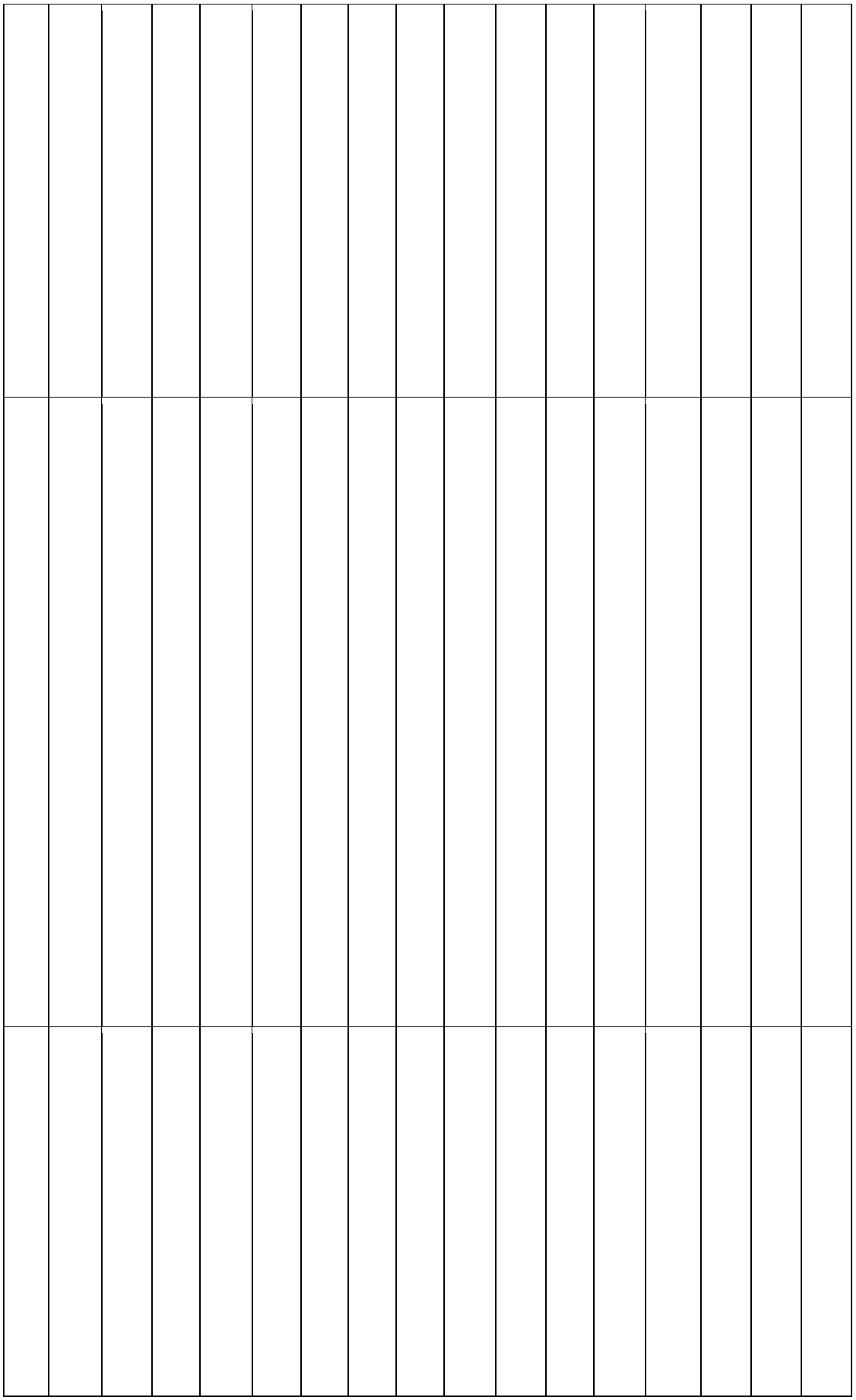
In [11]:
```
column_name = "Pclass"
df = train
dummies = pd.get_dummies(df[column_name],prefix=column_name)
dummies.head()
```
Out[11]:

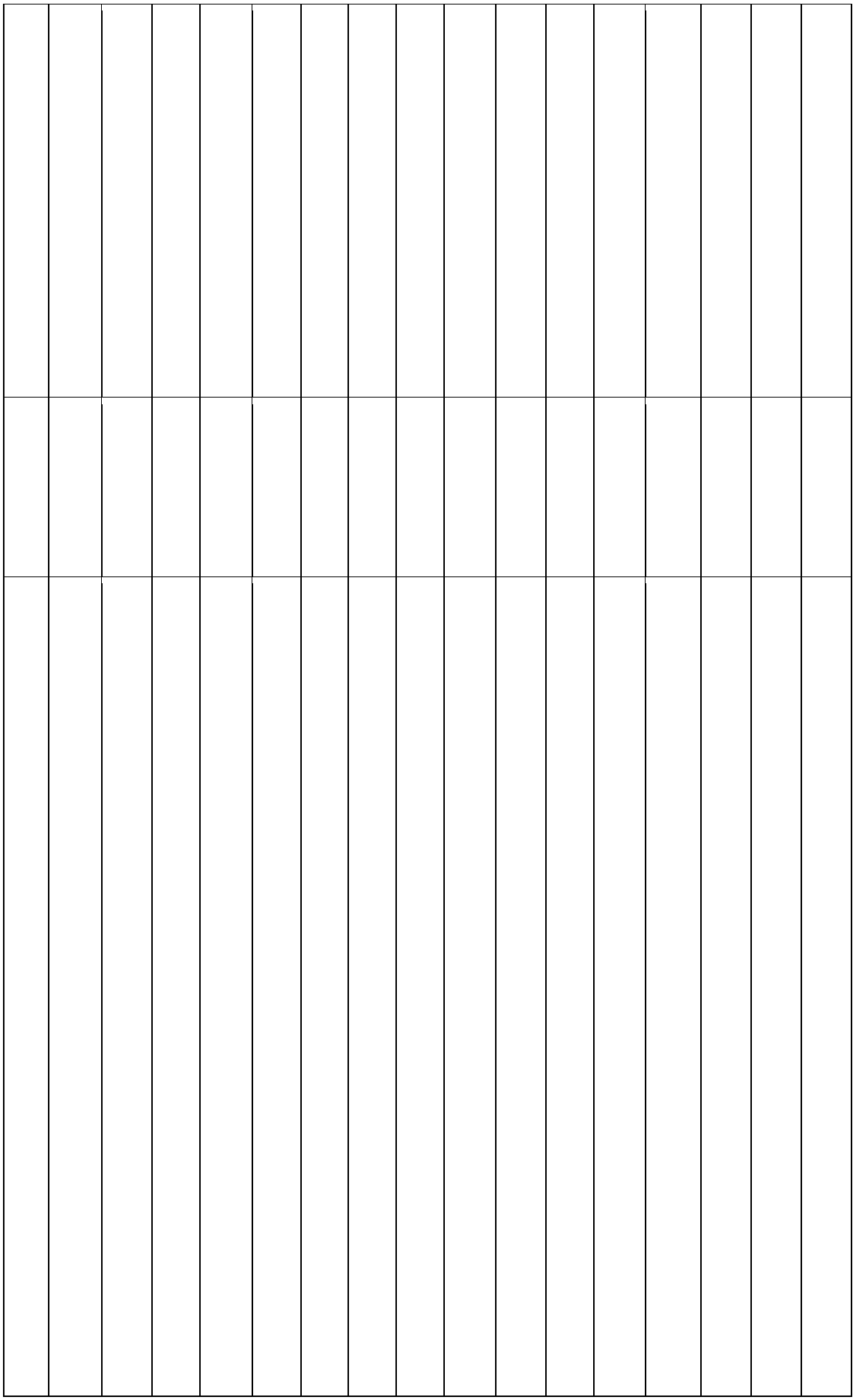
	Pclass_1	Pclass_2	Pclass_3
0	0	0	1
1	1	0	0
2	0	0	1
3	1	0	0
4	0	0	1

In [12]:
```
def create_dummies(df,column_name):
    dummies = pd.get_dummies(df[column_name],prefix=column_name)
    df = pd.concat([df,dummies],axis=1)
    return df

train = create_dummies(train,"Pclass")
test = create_dummies(test,"Pclass")
train.head()
```
Out[12]:







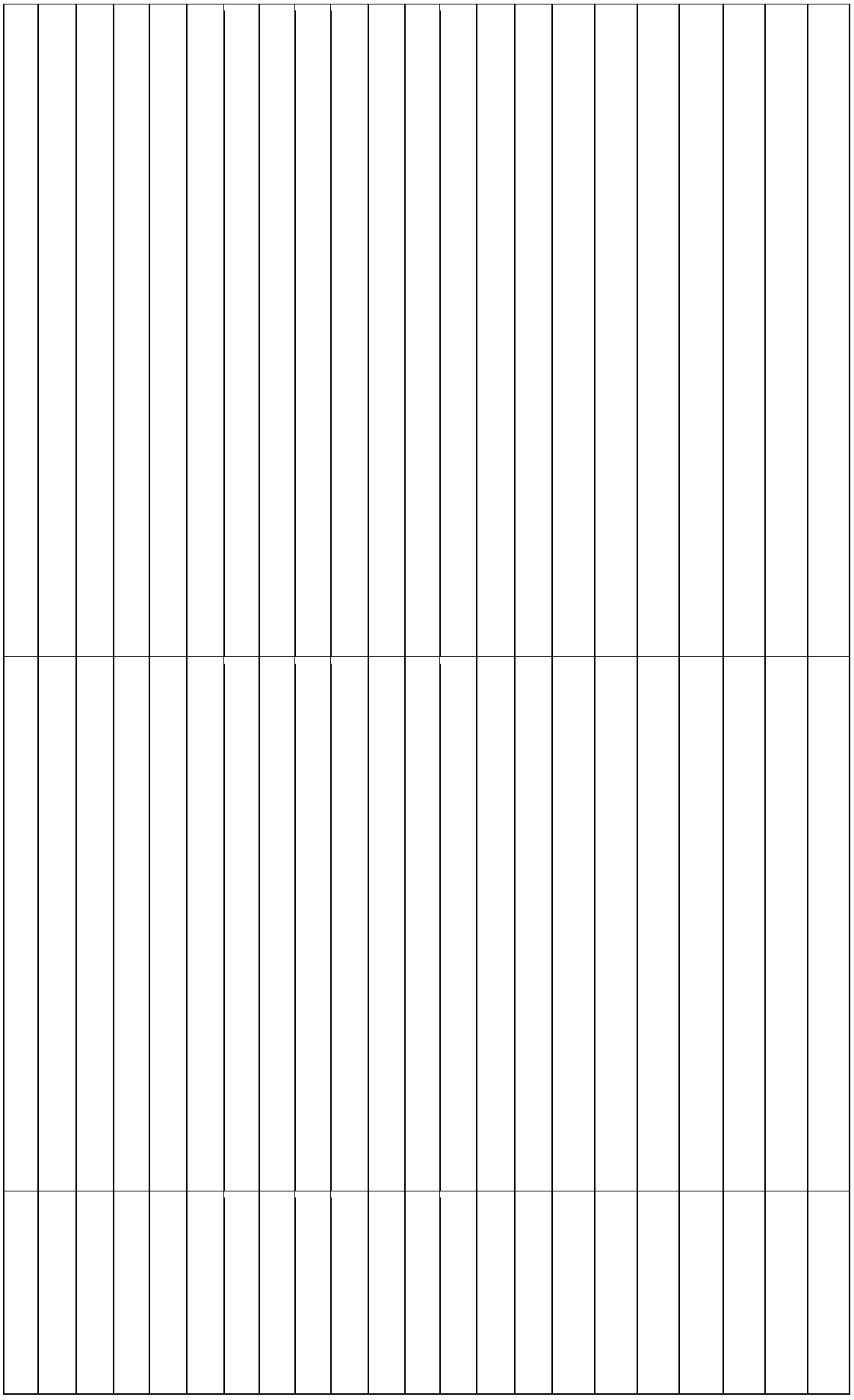
In [13]:

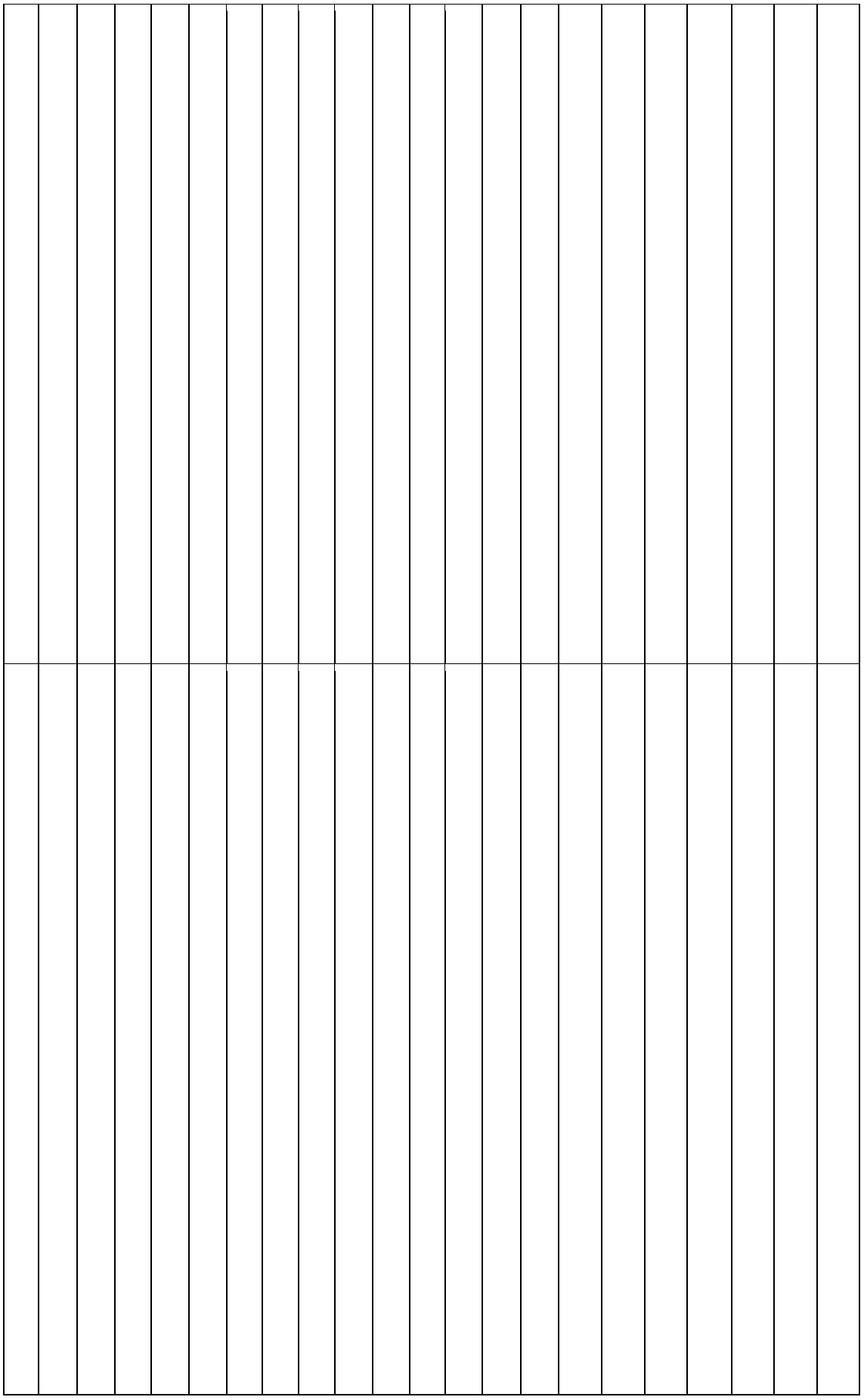
```
train = create_dummies(train,"Sex")
test = create_dummies(test,"Sex")
train = create_dummies(train,"Age_categories")
test = create_dummies(test,"Age_categories")
```

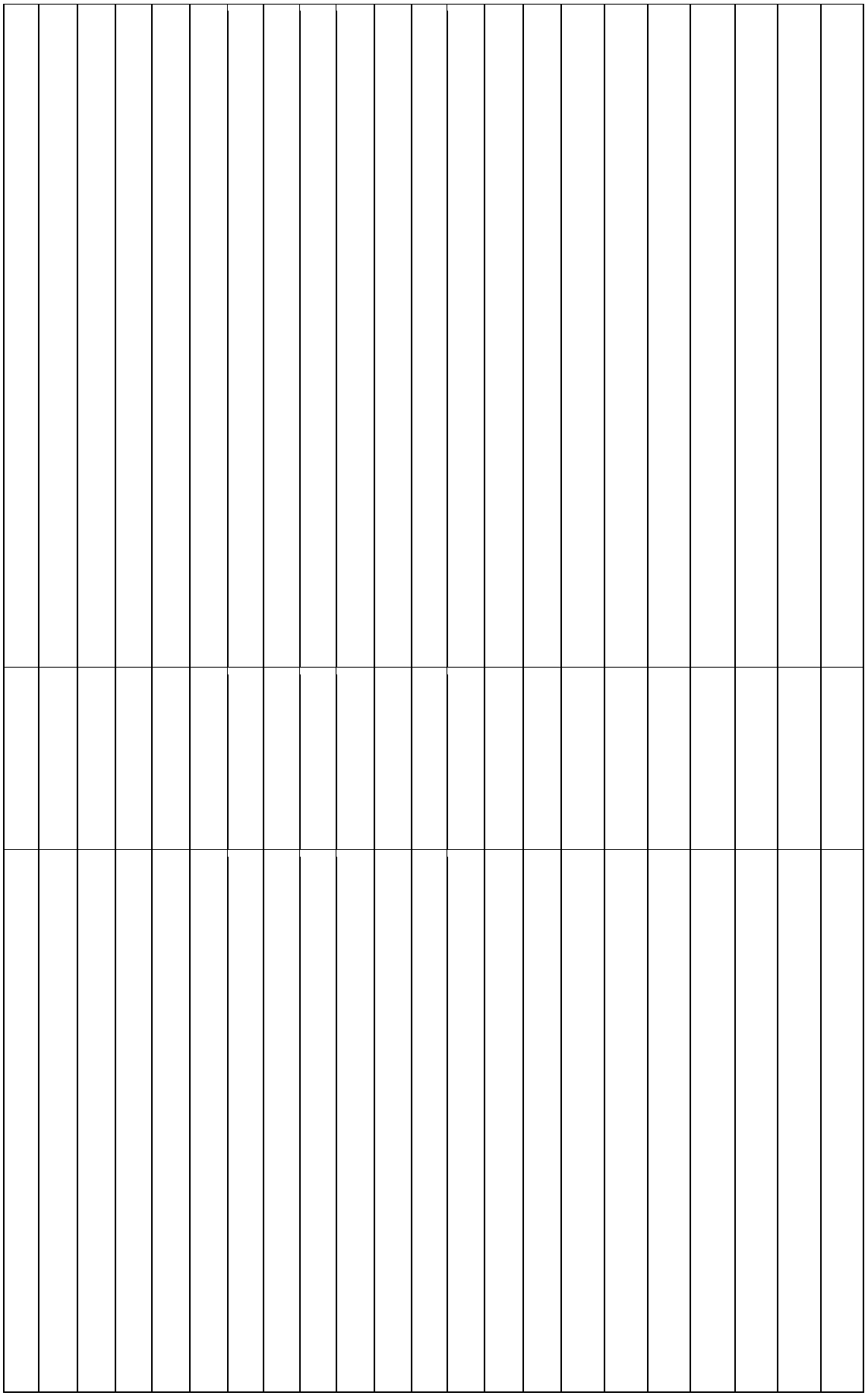
In [14]:

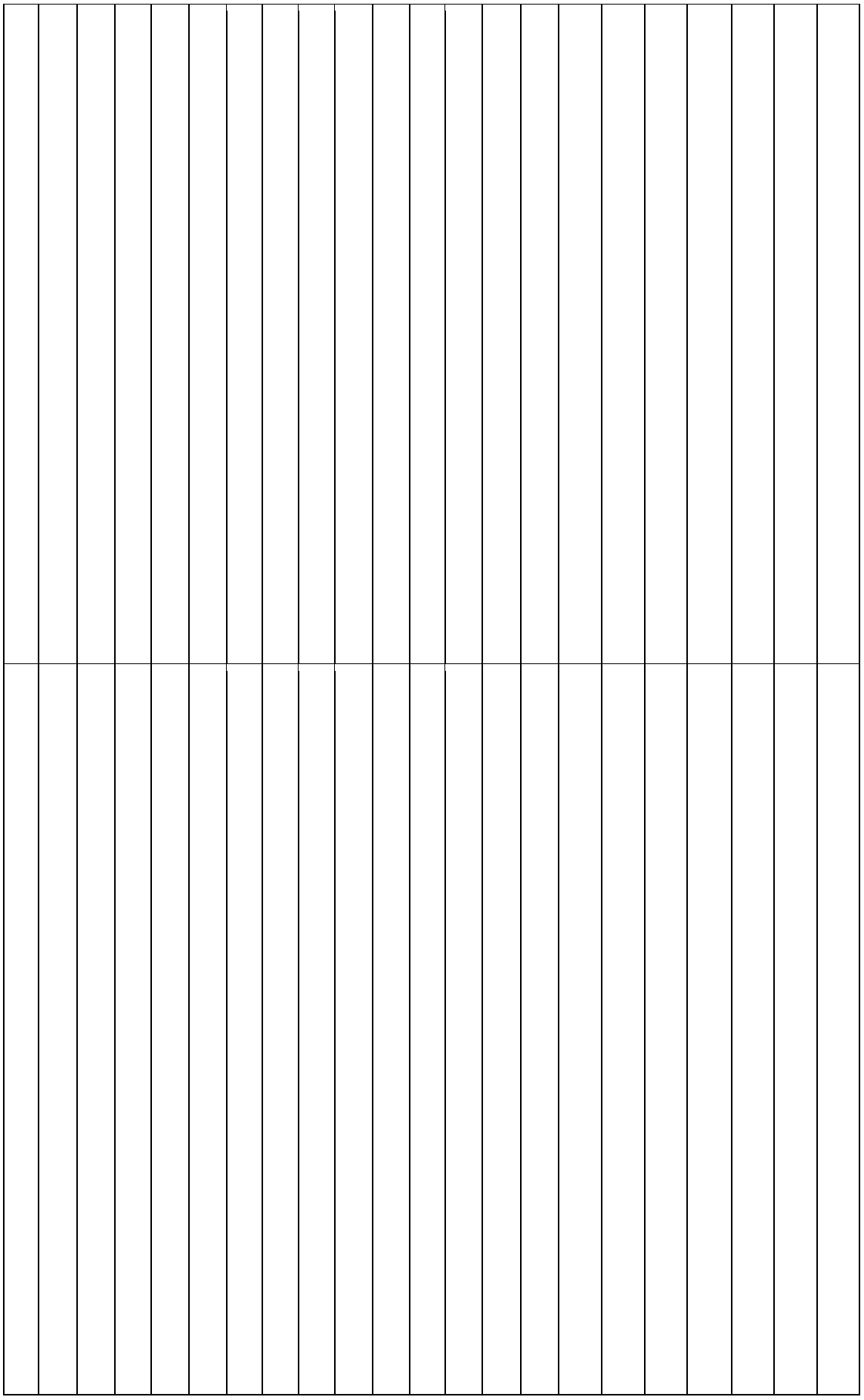
```
train.head()
```

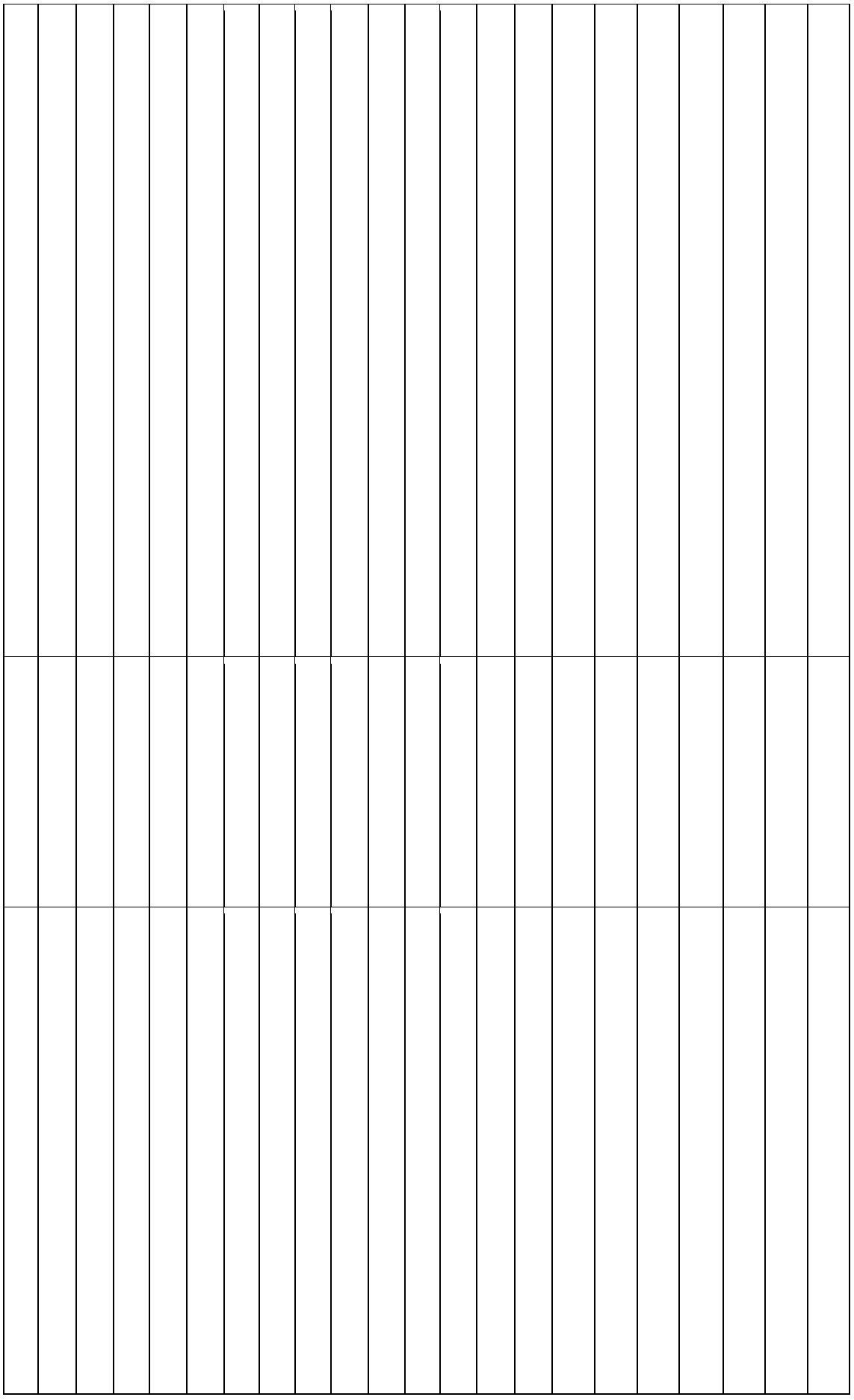
Out[14]:











5 rows × 25 columns

5. Creating our first machine learning model

Now that our data has been prepared, we are ready to train our first model. The first model we will use is called Logistic Regression, which is often the first model you will train when performing classification.

We will be using the scikit-learn library as it has many tools that make performing machine learning easier. The scikit-learn workflow consists of four main steps:

- Instantiate (or create) the specific machine learning model you want to use
- Fit the model to the training data
- Use the model to make predictions
- Evaluate the accuracy of the predictions
- Each model in scikit-learn is implemented as a separate class and the first step is to identify the class we want to create an instance of. In our case, we want to use the LogisticRegression class.

We'll start by looking at the first two steps. First, we need to import the class:

```
from sklearn.linear_model import LogisticRegression
```

Next, we create a LogisticRegression object:

```
lr = LogisticRegression()
```

Lastly, we use the `LogisticRegression.fit()` method to train our model.
The `.fit()` method accepts two arguments: X and y. X must be a two dimensional array (like a dataframe) of the features that we wish to train our model on, and y must be a one-dimensional array (like a series) of our target, or the column we wish to predict.

```
columns = ['Pclass_2', 'Pclass_3', 'Sex_male']

lr.fit(train[columns], train['Survived'])
```

The code above fits (or trains) our LogisticRegression model using three columns: Pclass_2, Pclass_3, and Sex_male.

Let's train our model using all of the columns we created above.

Instructions

- Instantiate a LogisticRegression object called lr.
- Use LogisticRegression.fit() to fit the model on the train dataset using:
    - The columns contained in columns as the first (X) parameter.
    - The Survived column as the second (y) parameter.

```
In [15]:
columns = ['Pclass_1', 'Pclass_2', 'Pclass_3', 'Sex_female', 'Sex_male',
       'Age_categories_Missing','Age_categories_Infant',
       'Age_categories_Child', 'Age_categories_Teenager',
       'Age_categories_Young Adult', 'Age_categories_Adult',
       'Age_categories_Senior']

from sklearn.linear_model import LogisticRegression
In [16]:

lr.decision_function(train[columns])
---------------------------------------------------------------------------
--
NameError                                 Traceback (most recent call las
t)
<ipython-input-16-5191cd5c5177> in <module>()
----> 1 lr.decision_function(train[columns])

NameError: name 'lr' is not defined
In [17]:

lr.coef_
---------------------------------------------------------------------------
--
NameError                                 Traceback (most recent call las
t)
<ipython-input-17-00d1c82c361d> in <module>()
----> 1 lr.coef_

NameError: name 'lr' is not defined
In [18]:

lr = LogisticRegression()
lr.fit(train[columns], train['Survived'])
Out[18]:
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tr
ue,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001
,
          verbose=0, warm_start=False)
```

6. Splitting our Training Data

Congratulations, you've trained your first machine learning model! Our next step is to find out how accurate our model is, and to do that, we'll have to make some predictions.

If you recall from earlier, we do have a test dataframe that we could use to make predictions. We could make predictions on that data set, but because it doesn't have the Survived column to find out our accuracy. This would quickly become a pain if we had to submit to find out the accuracy every time we optimized our model.

We could also fit and predict on our train dataframe, however if we do this there is a high likelihood that our model will overfit, which means it will perform well because we're testing on the same data we've trained on, but then perform much worse on new, unseen data.

Instead we can split our train dataframe into two:

- One part to train our model on (often 80% of the observations)
- One part to make predictions with and test our model (often 20% of the observations)

The convention in machine learning is to call these two parts train and test. This can become confusing, since we already have our test dataframe that we will eventually use to make predictions. To avoid confusion, from here on, we're going to call 'test' data holdout data, which is the technical name given to this type of data used for final predictions.

The scikit-learn library has a handy `model_selection.train_test_split()` function that we can use to split our data. train_test_split() accepts two parameters, `X` and `y`, which contain all the data we want to train and test on, and returns four objects: `train_X, train_y, test_X, test_y`:

Here's what the syntax for creating these four objects looks like:

```
from sklearn.model_selection import train_test_split



columns = ['Pclass_2', 'Pclass_3', 'Sex_male']



all_X = train[columns]

all_y = train['Survived']



train_X, test_X, train_y, test_y = train_test_split(

    all_X, all_y, test_size=0.2,random_state=0)
```

You'll notice that there are two other parameters we used: `test_size`, which lets us control what proportions our data are split into, and `random_state`.
The `train_test_split()` function randomizes observations before dividing them, and setting a random seed means that our results will be reproducible, which is important if you are collaborating, or need to produce consistent results each time.

Instructions

- Use the model_selection.train_test_split() function to split the train dataframe using the following parameters:
    - test_size of 0.2.
    - random_state of 0.
- Assign the four returned objects to train_X, test_X, train_y, and test_y.

In [19]:
```
holdout = test # from now on we will refer to this
               # dataframe as the holdout data

from sklearn.model_selection import train_test_split

columns = ['Pclass_1', 'Pclass_2', 'Pclass_3', 'Sex_female', 'Sex_male',
       'Age_categories_Missing','Age_categories_Infant',
       'Age_categories_Child', 'Age_categories_Teenager',
       'Age_categories_Young Adult', 'Age_categories_Adult',
       'Age_categories_Senior']

all_X = train[columns]
all_y = train['Survived']

train_X, test_X, train_y, test_y = train_test_split(
    all_X, all_y, test_size=0.2,random_state=0)
```

In [20]:
```
train_X.shape
```

Out[20]:

```
(712, 12)
```

7. Making Predictions and Measuring their Accuracy

Now that we have our data split into train and test sets, we can fit our model again on our training set, and then use that model to make predictions on our test set.

Once we have fit our model, we can use the `LogisticRegression.predict()` method to make predictions.

The `predict()` method takes a single parameter `X`, a two dimensional array of features for the observations we wish to predict. `X` must have the exact same features as the array we used to fit our model. The method returns single dimensional array of predictions.

```
lr = LogisticRegression()

lr.fit(train_X, train_y)

predictions = lr.predict(test_X)
```

There are a number of ways to measure the accuracy of machine learning models.

In this case, our score calculated as "the percentage of passengers correctly predicted". This is by far the most common form of accuracy for binary classification.

As an example, imagine we were predicting a small data set of five observations.

Our model's prediction	The actual value	Correct
0	0	Yes
1	0	No
0	1	No
1	1	Yes
1	1	Yes

In this case, our model correctly predicted three out of five values, so the accuracy based on this prediction set would be 60%.

Again, scikit-learn has a handy function we can use to calculate accuracy: `metrics.accuracy_score()`. The function accepts two parameters, `y_true` and `y_pred`, which are the actual values and our predicted values respectively, and returns our accuracy score.

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(test_y, predictions)
```

Let's put all of these steps together, and get our first accuracy score.

Instructions

- Instantiate a new `LogisticRegression()` object, `lr`.
- Fit the model using `train_X` and `train_y`.
- Make predictions using `test_X` and assign the results to `predictions`.
- Use `accuracy_score()` to compare `test_y` and `predictions`, assigning the result to `accuracy`
- Print the `accuracy` variable.

In [21]:
```
from sklearn.metrics import accuracy_score
```
In [22]:
```
lr = LogisticRegression()
lr.fit(train_X, train_y)
```

```
predictions = lr.predict(test_X)
accuracy = accuracy_score(test_y, predictions)
accuracy
```

Out[22]:

```
0.81005586592178769
```

In [23]:

```
from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(test_y, predictions)
pd.DataFrame(conf_matrix, columns=['Survived', 'Died'], index=[['Survived', '
Died']])
```

Out[23]:

	Survived	Died
Survived	96	14
Died	20	49

Using Cross Validation for More Accurate Error Measurement

Our model has an accuracy score of 81.0% when tested against our 20% test set. Given that this data set is quite small, there is a good chance that our model is overfitting, and will not perform as well on totally unseen data.

To give us a better understanding of the real performance of our model, we can use a technique called **cross validation** to train and test our model on different splits of our data, and then average the accuracy scores.

The most common form of cross validation, and the one we will be using, is called **k-fold** cross validation. 'Fold' refers to each different iteration that we train our model on, and 'k' just refers to the number of folds. In the diagram above, we have illustrated k-fold validation where k is 5.

We will use scikit-learn's `model_selection.cross_val_score()` function to automate the process. The basic syntax for `cross_val_score()` is:

```
cross_val_score(estimator, X, y, cv=None)
```

- estimator is a scikit-learn estimator object, like the LogisticRegression() objects we have been creating.
- X is all features from our data set.
- y is the target variables.
- cv specifies the number of folds.

The function returns a numpy ndarray of the accuracy scores of each fold.

It's worth noting, the `cross_val_score()` function can use a variety of cross validation techniques and scoring types, but it defaults to k-fold validation and accuracy scores for our input types.

Instructions

- Instantiate a new LogisticRegression() object, lr.
- Use model_selection.cross_val_score() to perform cross-validation on our data and assign the results to scores:
    - Use the newly created lr as the estimator.
    - Use all_X and all_y as the input data.
    - Specify 10 folds to be used.
- Use the numpy.mean() function to calculate the mean of scores and assign the result to accuracy.
- Print the variables scores and accuracy.

In [24]:
```
from sklearn.model_selection import cross_val_score
import numpy as np

lr = LogisticRegression()
scores = cross_val_score(lr, all_X, all_y, cv=10)
np.mean(scores)
```

Out[24]:

```
0.80246708659630017
```

9. Making Predictions on Unseen Data

From the results of our k-fold validation, you can see that the accuracy number varies with each fold - ranging between 76.4% and 87.6%. This demonstrates why cross validation is important.

As it happens, our average accuracy score was 80.2%, which is not far from the 81.0% we got from our simple train/test split, however this will not always be the case, and you should always use cross-validation to make sure the error metrics you are getting from your model are accurate.

We are now ready to use the model we have built to train our final model and then make predictions on our unseen holdout data, or the 'test' data set.

Instructions

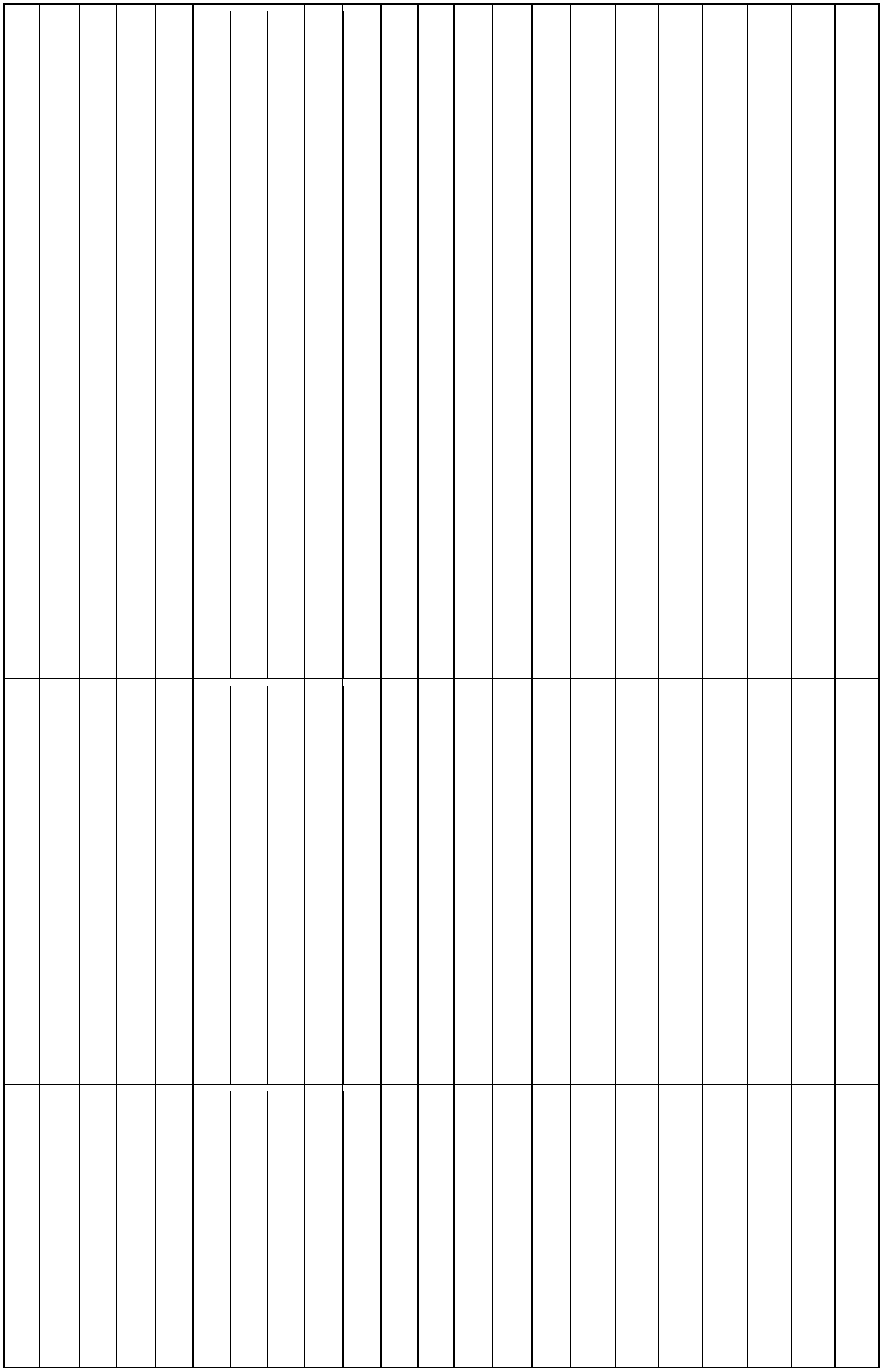
- Instantiate a new `LogisticRegression()` object, `lr`.
- Use the `fit()` method to train the model `lr` using training data: `all_X` and `all_y`.
- Make predictions using the `holdout` data and assign the result to `holdout_predictions`.

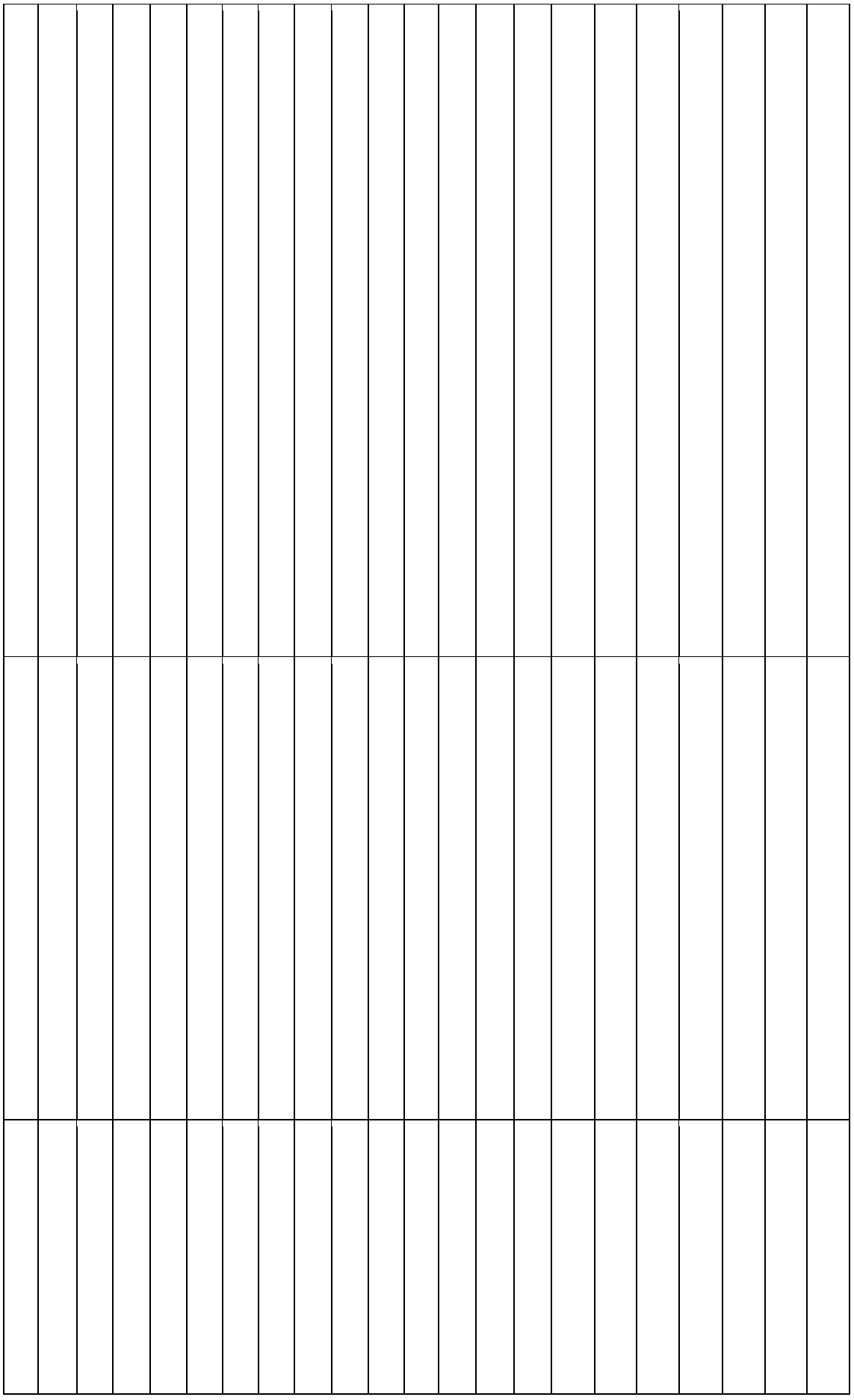
In [25]:
```
columns = ['Pclass_1', 'Pclass_2', 'Pclass_3', 'Sex_female', 'Sex_male',
       'Age_categories_Missing','Age_categories_Infant',
       'Age_categories_Child', 'Age_categories_Teenager',
       'Age_categories_Young Adult', 'Age_categories_Adult',
       'Age_categories_Senior']
```

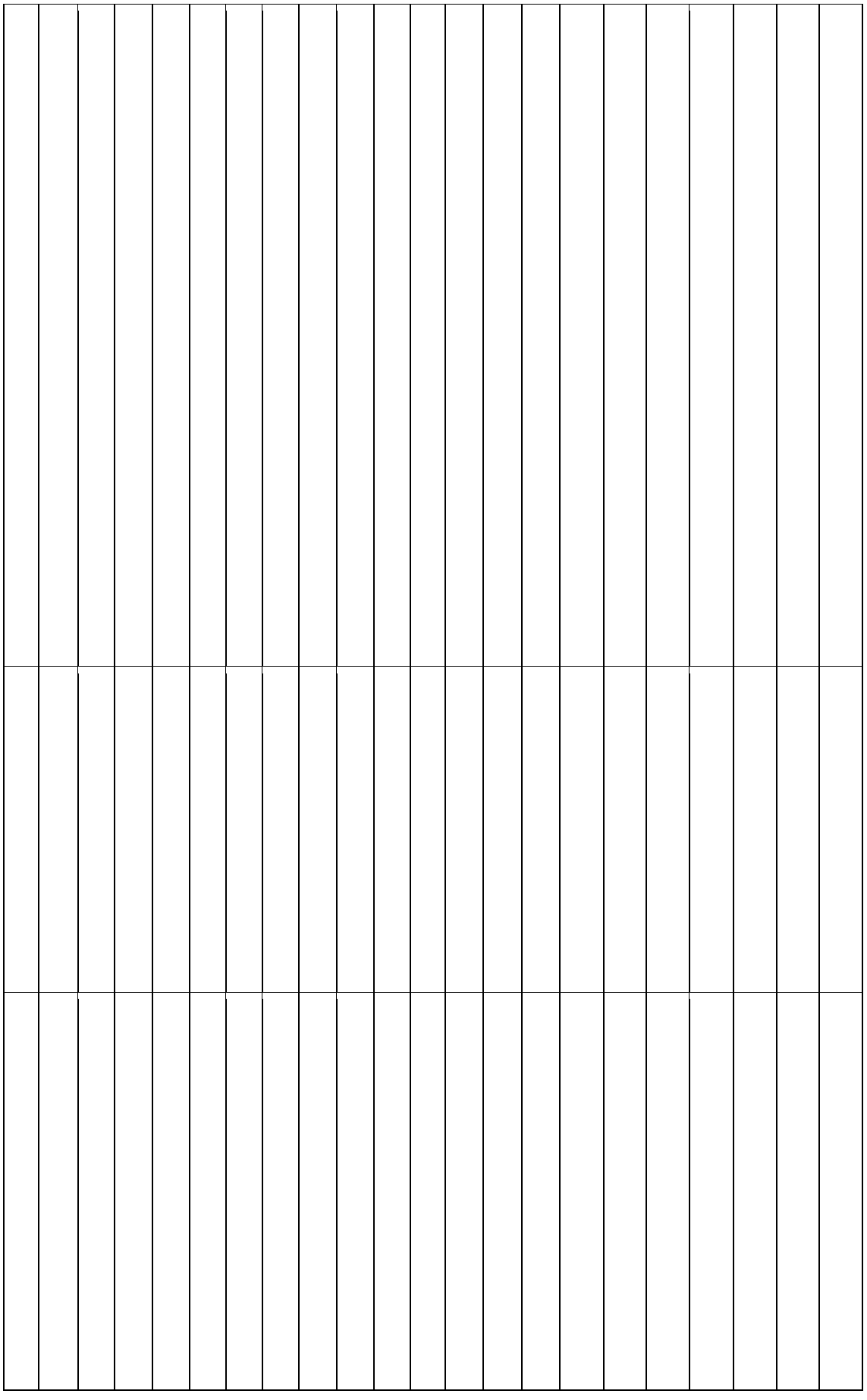
In [26]:

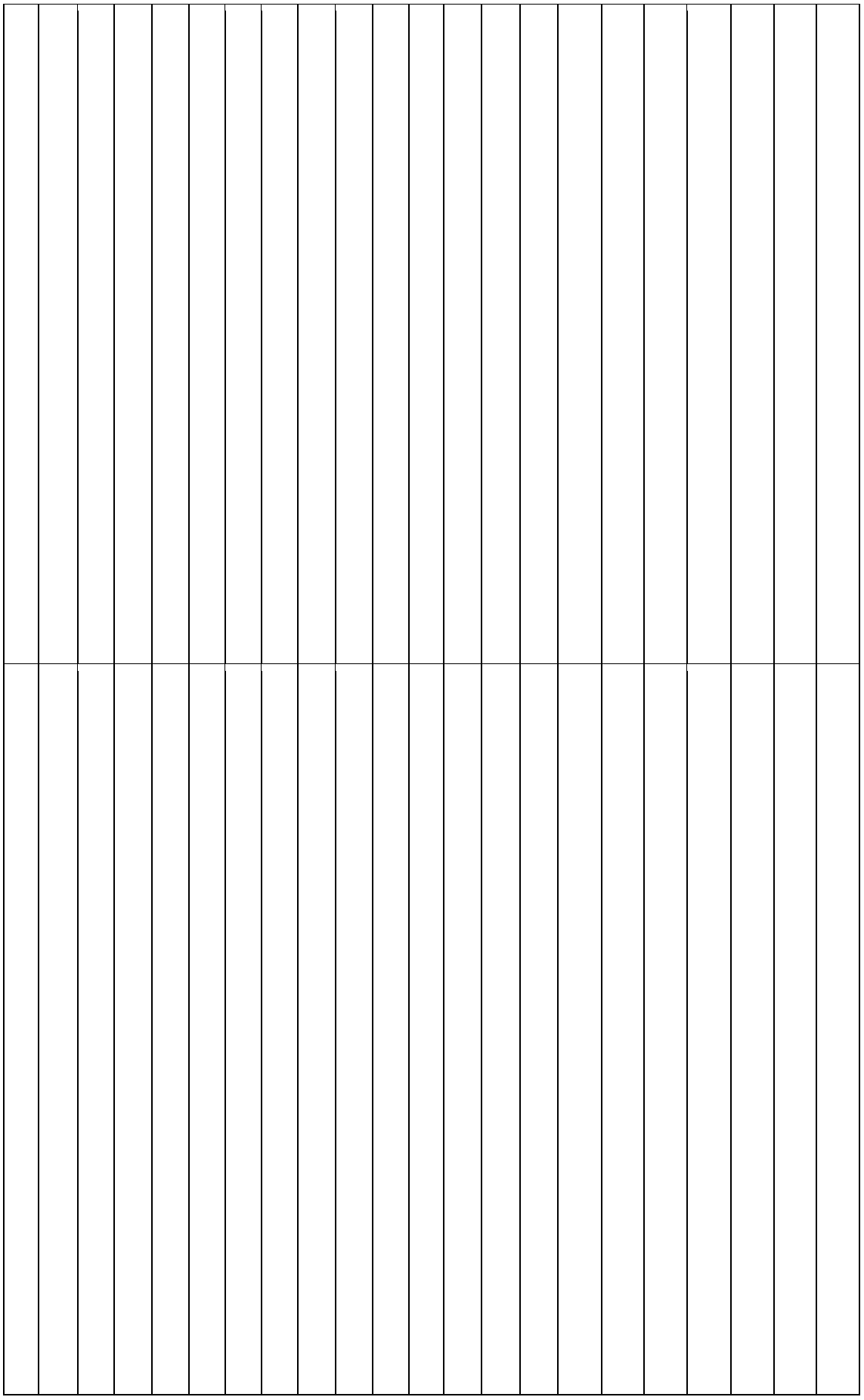
```
holdout.head()
```

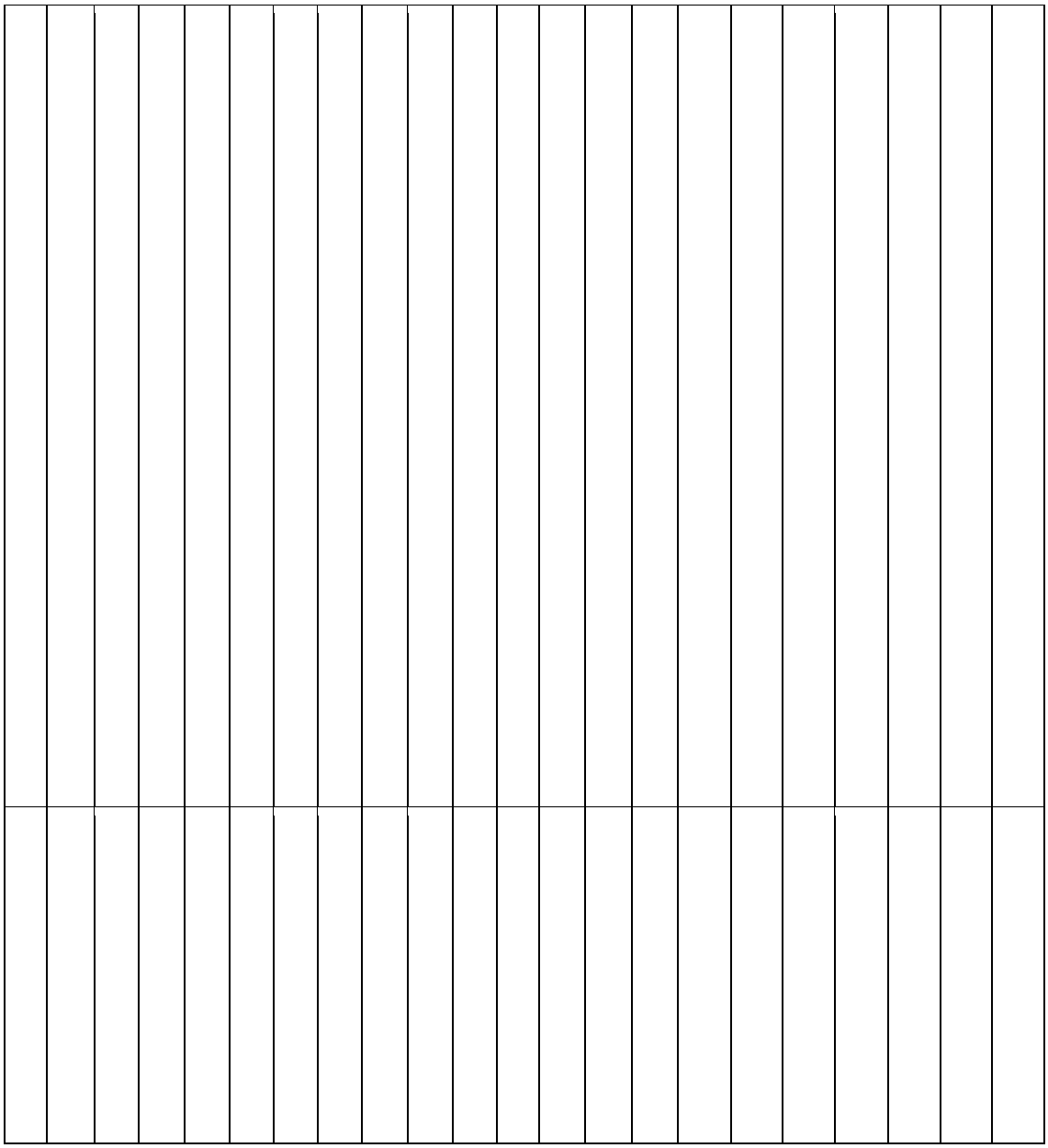
Out[26]:











5 rows × 24 columns

In [27]:

```
lr = LogisticRegression()
lr.fit(all_X, all_y)
holdout_predictions = lr.predict(holdout[columns])
holdout_predictions
```

Out[27]:

```
array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
1,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1,
0,
       0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0,
0,
       1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
0,
```

```
        1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
1,
        0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
0,
        1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1,
1,
        0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1,
0,
        1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0,
1,
        0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0,
0,
        0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0,
0,
        0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
1,
        0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
0,
        0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
0,
        0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0,
        1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
1,
        1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
0,
        1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
0,
        1, 0, 0, 0]
```