Model

We experimented different setups using the findings from the individual experiments to decide the final model. F1 score was used as a metric because the text should not be classified to wrong classes or miss important classes.

From the results of the individual experiments, we discovered that the deep learning approach is superior to the shallow learning approach. Therefore we decided to choose one model from LSTM, GRU and CNN. As having many classes is better for our case, we experimented with a combination of all levels of classes. It gave a reasonable accuracy so we decided to use all levels for the final model.

Following setups were experimented to choose the final model.

- Balancing data
  - Balanced
  - ***Not balanced***
- Preprocessing
  - Tokenize
  - ***Remove punctuation, remove stopwords***
  - Filter, remove stopwords
- Model
  - LSTM
  - ***GRU***
  - CNN
- Optimizers
  - ***RMSprop***
  - Adam
- Activation function
  - ***Sigmoid***
  - Tanh

The best setup found from the experiments are shown in bold.

In [2]:

```
import pandas as pd
import numpy as np
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
import string
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, SpatialDropout1D, GRU, GlobalA
from sklearn.preprocessing import MultiLabelBinarizer
!pip install tensorflow_addons
import tensorflow_addons as tfa
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import f1_score, roc_auc_score
from joblib import dump
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\rina9\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Requirement already satisfied: tensorflow_addons in c:\users\rina9\anaconda3\envs\co
m3029\lib\site-packages (0.12.1)
Requirement already satisfied: typeguard>=2.7 in c:\users\rina9\anaconda3\envs\com30
29\lib\site-packages (from tensorflow_addons) (2.12.0)
```

In [3]:
```
train = pd.read_csv('./data/DBPEDIA_train.csv')
test = pd.read_csv('./data/DBPEDIA_test.csv')
val = pd.read_csv('./data/DBPEDIA_val.csv')
```

In [4]:
```
train['Classes'] = train['l1'] + " " + train['l2'] + " " + train['l3']
test['Classes'] = test['l1'] + " " + test['l2'] + " " + test['l3']
val['Classes'] = val['l1'] + " " + val['l2'] + " " + val['l3']
```

In [5]:
```
train = train.drop(['l1','l2','l3'], axis=1)
test = test.drop(['l1','l2','l3'], axis=1)
val = val.drop(['l1','l2','l3'], axis=1)
```

In [6]:
```
train['Classes'] = train['Classes'].str.split()
test['Classes'] = test['Classes'].str.split()
val['Classes'] = val['Classes'].str.split()
```

In [7]:
```
mlb = MultiLabelBinarizer()
train_y = mlb.fit_transform(train['Classes'])
test_y = mlb.transform(test['Classes'])
val_y = mlb.transform(val['Classes'])
```

In [8]:
```
dump(mlb, 'mlb.joblib')
```

Out[8]:
```
['mlb.joblib']
```

In [ ]:
```
# Remove PUNCTUATION
def remove_punctuation(text):
  table = str.maketrans("","", string.punctuation)
  return text.translate(table)

# Remove STOPWORDS & LOWERCASE
stops = set(stopwords.words("english"))
def remove_stopwords(text):
  text = [word.lower() for word in text.split() if word.lower() not in stops and wor
  return " ".join(text)
```

In [ ]:
```
train['text'] = train.text.map(lambda x: remove_punctuation(x))
train['text'] = train['text'].map(remove_stopwords)
test['text'] = test.text.map(lambda x: remove_punctuation(x))
test['text'] = test['text'].map(remove_stopwords)
val['text'] = val.text.map(lambda x: remove_punctuation(x))
val['text'] = val['text'].map(remove_stopwords)
```

In [ ]:
```
tokenizer = Tokenizer(num_words=250000)
tokenizer.fit_on_texts(train.text)
```

```
train_data = tokenizer.texts_to_sequences(train.text)
test_data = tokenizer.texts_to_sequences(test.text)
val_data = tokenizer.texts_to_sequences(val.text)
word_index = tokenizer.word_index
```

In [22]:
```
max_words = 250000
max_length = 500
```

In [ ]:
```
train_data = pad_sequences(train_data, maxlen=max_length, padding="post", truncating
test_data = pad_sequences(test_data, maxlen=max_length, padding="post", truncating="
val_data = pad_sequences(val_data, maxlen=max_length, padding="post", truncating="po
```

After removing punctuations and stopwords, the texts are tokenized with the maximum number of words set to 250,000, which is about a half of the total number of words in the dataset. Tokenized texts are then transformed to sequences of integers and padded to the length of 500. The classes are one-hot encoded.

In [23]:
```
def get_model(embedding_dim, activation, loss, optimizer):

  model = Sequential()
  model.add(Embedding(max_words, embedding_dim, input_length=max_length))
  model.add(SpatialDropout1D(0.3))
  model.add(Bidirectional(GRU(100, return_sequences=True)))
  model.add(GlobalAveragePooling1D())
  model.add(Dense(298, activation='sigmoid'))

  model.compile(loss=loss, optimizer=optimizer, metrics=[tfa.metrics.F1Score(298, 'm
  return model
```

In [24]:
```
model = get_model(embedding_dim=512, loss='binary_crossentropy', optimizer='RMSprop'
```

In [ ]:
```
history = model.fit(train_data, train_y, epochs=10, batch_size=64, validation_data=(
```

```
Epoch 1/10
3765/3765 [==============================] - 1822s 475ms/step - loss: 0.0389 - f1_sc
ore: 0.3723 - val_loss: 0.0072 - val_f1_score: 0.4880
Epoch 2/10
3765/3765 [==============================] - 1816s 482ms/step - loss: 0.0058 - f1_sc
ore: 0.4902 - val_loss: 0.0038 - val_f1_score: 0.4923
Epoch 3/10
3765/3765 [==============================] - 1804s 479ms/step - loss: 0.0032 - f1_sc
ore: 0.4941 - val_loss: 0.0031 - val_f1_score: 0.4936
Epoch 4/10
3765/3765 [==============================] - 1804s 479ms/step - loss: 0.0024 - f1_sc
ore: 0.4956 - val_loss: 0.0028 - val_f1_score: 0.4940
Epoch 5/10
3765/3765 [==============================] - 1806s 480ms/step - loss: 0.0019 - f1_sc
ore: 0.4967 - val_loss: 0.0027 - val_f1_score: 0.4944
Epoch 6/10
3765/3765 [==============================] - 1787s 475ms/step - loss: 0.0015 - f1_sc
ore: 0.4978 - val_loss: 0.0027 - val_f1_score: 0.4949
Epoch 7/10
3765/3765 [==============================] - 1790s 476ms/step - loss: 0.0012 - f1_sc
ore: 0.4982 - val_loss: 0.0028 - val_f1_score: 0.4948
Epoch 8/10
3765/3765 [==============================] - 1792s 476ms/step - loss: 9.9089e-04 - f
1_score: 0.4988 - val_loss: 0.0029 - val_f1_score: 0.4949
```

```
Epoch 9/10
3765/3765 [==============================] - 1788s 475ms/step - loss: 8.4480e-04 - f
1_score: 0.4990 - val_loss: 0.0030 - val_f1_score: 0.4951
Epoch 10/10
3765/3765 [==============================] - 1783s 474ms/step - loss: 6.8457e-04 - f
1_score: 0.4995 - val_loss: 0.0032 - val_f1_score: 0.4953
```

The final model was trained for 10 epochs with a batch size of 64. The calculation of the f1 score using the tensorflow_addons package did not perform as expected so we decided to use the function in the scikit-learn package instead.

In [ ]:
```
preds = model.predict(test_data)
```

In [ ]:
```
y_preds = np.where(preds < 0.9, 0, 1)
y_true = test_y
f1 = f1_score(y_true, y_preds, average='micro')*100
print("Test result: %0.1f%%" % f1)
roc_auc = roc_auc_score(y_true, preds, 'micro') * 100
print("ROC AUC score: %0.1f%%" % roc_auc)
```

```
Test result: 95.8%
ROC AUC score: 99.9%
```

In [25]:
```
name = "first"
```

In [26]:
```
filename = "model_{}".format(name)
model.save(filename)
```

```
WARNING:absl:Found untraced functions such as gru_cell_7_layer_call_fn, gru_cell_7_l
ayer_call_and_return_conditional_losses, gru_cell_8_layer_call_fn, gru_cell_8_layer_
call_and_return_conditional_losses, gru_cell_7_layer_call_fn while saving (showing 5
of 10). These functions will not be directly callable after loading.
WARNING:absl:Found untraced functions such as gru_cell_7_layer_call_fn, gru_cell_7_l
ayer_call_and_return_conditional_losses, gru_cell_8_layer_call_fn, gru_cell_8_layer_
call_and_return_conditional_losses, gru_cell_7_layer_call_fn while saving (showing 5
of 10). These functions will not be directly callable after loading.
INFO:tensorflow:Assets written to: model_first\assets
INFO:tensorflow:Assets written to: model_first\assets
```

Task 1: Model Serving Options

There are various model serving options that can be used for deploying our model as a web service.

Flask is a lightweight, micro web framework built in Python to deploy web applications. It can be used with Gunicorn and Nginx. Flask works as an application server which communicates with the model. Gunicorn gives the ability to build the web service through HTTP server and Nginx setup the environment on a specific port.

TensorFlow Serving another option that can be used for web serving. It handles minibatching and is optimized for speed as it can use GPUs. It is specifically made for models so the latency is much smaller.

TorchServe provides metrics for monitoring, RESTful endpoints for application integration and supports ML environments such as Amazon SageMaker, etc. It is flexible and easy-to-use but it

can only be used for serving PyTorch models.

Django is a full-stack web framework built in Python to deploy web applications. It is better for building full-featured web applications with lots of functionalities if the required ML service is simple. For the same functionality, Django needs 2x more lines of code than Flask.

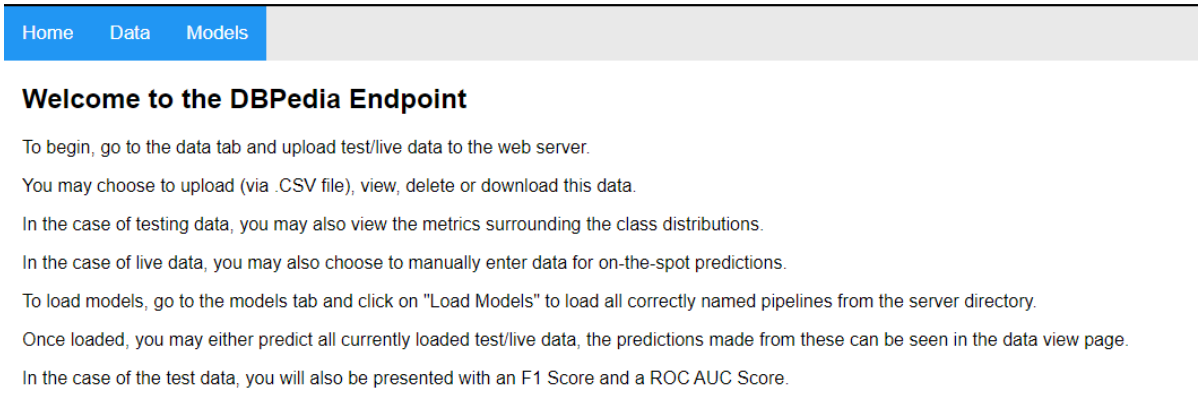
Although Flask has the bare minimum for a web server, it provides everything required for building a basic web service. For our case, using Flask for web application with the Waitress WSGI to make it into a web service, would be sufficient.

Task 2: Web Service

We developed a web application (server.py) that allows a user to test classified articles and to classify new articles. There are three tabs on the web page: Home, Data and Models.

Home

This tab is the first tab that is displayed when the user access to the web page.



Data

This tab allows the user to add, remove and view data. There are two types of data: Testing data and Live data. Both are segregated into 2 different storage devices. Testing data is used for the purpose of evaluating and collecting performance metrics from the endpoint models. Whereas live data represents data that is being used exclusively for the purpose of prediction.

The class used to encapsulate the storage and basic interactions of the data is defined here:

In [1]:
```
class Storage:
    def __init__(self):
        self.list = []
        self.count = 0
        self.mlb = MultiLabelBinarizer()

    def load_file(self, filename, isLive = False):
        data = pd.read_csv(filename)
        texts = data['text']
        if not isLive:
            data['Classes'] = data['l1'] + " " + data['l2'] + " " + data['l3']
            data = data.drop(['l1','l2','l3'], axis=1)
```

```
            data['Classes'] = data['Classes'].str.split()

            display_labels = data['Classes'].to_numpy()
            labels = self.mlb.fit_transform(data['Classes'])
            print("labels", labels.shape)
            for i in range(len(texts)):
                self.add(texts[i], labels[i], display_labels[i])

        else:
            for i in range(len(texts)):
                self.add(texts[i], None, None)
        print("Loaded Data")

    def add(self, text, label, display_label):
        self.count += 1
        self.list.append(Article(self.count, datetime.now(), text, label, display_la

    def fetch_all(self):
        if self.count == 0:
            return "Empty"
        output = []
        for article in self.list:
            output.append({'article_id': article.id,
                           'created_at': article.created_at,
                           'article_text': article.text,
                           'display_label': article.display_label,
                           'prediction': self.mlb.inverse_transform(np.expand_dims(a
                           'label': article.label})
        return pd.DataFrame.from_dict(output)
```

In this class, the data is contained within a single list attribute, with an integer counter keeping track of the cumulative number of data entries. This is also used to issue ID numbers. The "load_file" method takes in a CSV file and extracts the article text and each hierarchical label. These labels are then concatenated together and transformed into a multi-label format and are stored along with each text entry for test data.

For live data only the article text is stored, as the purpose of live data is to handle real-world predictions and not to test the performance of the model.

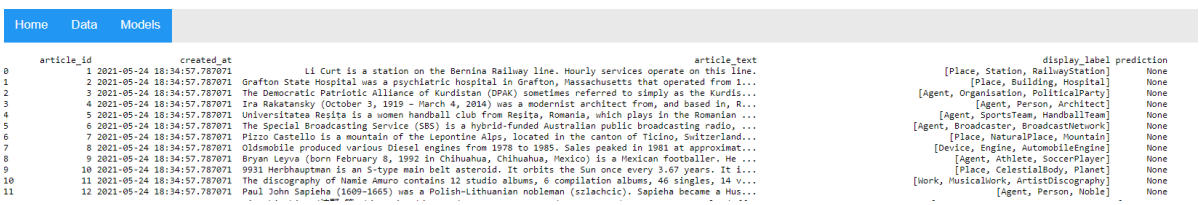
The "add" method appends a single record to the storage list. And the "fetch_all" method returns a dataframe complete with every record held within the current storage object.

Data Operations

The following is a set of actions the user can take to interact with the data storage:

View Data

The user can view the loaded data as a table. The table contains the data IDs, creation timestamps, article texts, expected labels and predicted labels of the wiki entries. The web page is difficult to load when it tries to load too much data so it is set to display maximum of 65000 entries.



	article_id	created_at	article_text	display_label	prediction
0	1	2021-05-24 18:34:57.787071	Li Curt is a station on the Bernina Railway line. Hourly services operate on this line.	[Place, Station, RailwayStation]	None
1	2	2021-05-24 18:34:57.787071	Grafton State Hospital was a psychiatric hospital in Grafton, Massachusetts that operated from 1...	[Place, Building, Hospital]	None
2	3	2021-05-24 18:34:57.787071	The Democratic Patriotic Alliance of Kurdistan (DPAK) sometimes referred to simply as the Kurdis...	[Agent, Organisation, PoliticalParty]	None
3	4	2021-05-24 18:34:57.787071	Ira Rakatansky (October 3, 1919 – March 4, 2014) was a modernist architect from, and based in, R...	[Agent, Person, Architect]	None
4	5	2021-05-24 18:34:57.787071	Universitatea Reșița is a women handball club from Reșița, Romania, which plays in the Romanian ...	[Agent, SportsTeam, HandballTeam]	None
5	6	2021-05-24 18:34:57.787071	The Special Broadcasting Service (SBS) is a hybrid-funded Australian public broadcasting radio, ...	[Agent, Broadcaster, BroadcastNetwork]	None
6	7	2021-05-24 18:34:57.787071	Pizzo Castello is a mountain of the Lepontine Alps, located in the canton of Ticino, Switzerland...	[Place, NaturalPlace, Mountain]	None
7	8	2021-05-24 18:34:57.787071	Oldsmobile produced various Diesel engines from 1978 to 1985. Sales peaked in 1981 at approximat...	[Device, Engine, AutomobileEngine]	None
8	9	2021-05-24 18:34:57.787071	Bryan Leyva (born February 8, 1992 in Chihuahua, Chihuahua, Mexico) is a Mexican footballer. He ...	[Agent, Athlete, SoccerPlayer]	None
9	10	2021-05-24 18:34:57.787071	9931 Herbhauptman is an S-type main belt asteroid. It orbits the Sun once every 3.67 years. It i...	[Place, CelestialBody, Planet]	None
10	11	2021-05-24 18:34:57.787071	The discography of Namie Amuro contains 12 studio albums, 6 compilation albums, 46 singles, 14 v...	[Work, MusicalWork, ArtistDiscography]	None
11	12	2021-05-24 18:34:57.787071	Paul John Sapieha (1609–1665) was a Polish-Lithuanian nobleman (szlachcic). Sapieha became a Hus...	[Agent, Person, Noble]	None

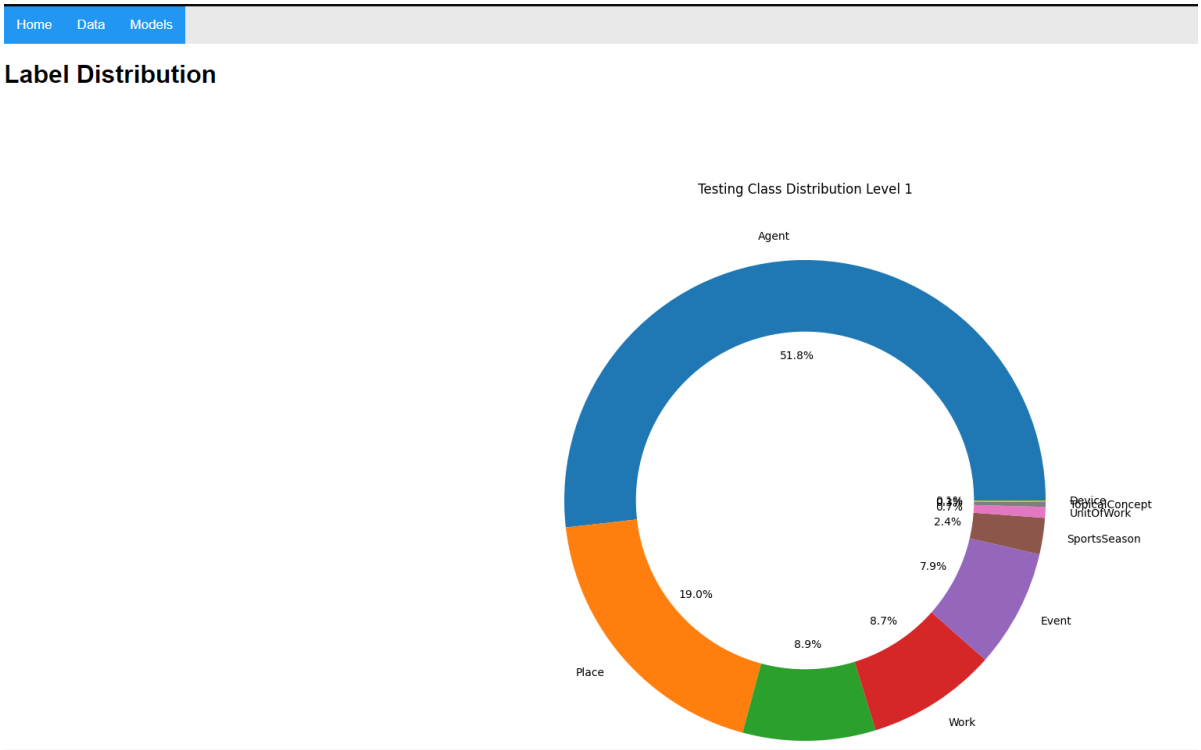
In [3]:

```
#Views data
#@app.route('/data/view', methods=['POST'])
def view_data():
    c = request.form['dataName']
    try:
        if c not in ["test", "live"]: return render_template('data.html', outcome="F
        if c == "test":
            table = test_storage.fetch_all()
        else:
            table = live_storage.fetch_all()
        del table['label']
    except Exception:
        return render_template('data.html', outcome="Storage Error")

    return render_template('data_view.html', table=table)
```

The flask request method to handle this is shown above. It first checks two exception cases: that the form is sending valid information in the form of the name of the storage being viewed, and that the storage objects exist. It then calls the "fetch_all()" method as defined previously to retain a dataframe to display the data.

View Metrics

This displays the class distributions in pie charts. There are three pie charts to display each level of classes.



In [1]:

```
def pieChart(name, lvl, labels):
    plt.figure(figsize=(20,10))
    circle=plt.Circle( (0,0), 0.7, color='white')
    plt.pie(labels, labels=labels.index, autopct='%1.1f%%')
    p=plt.gcf()
    p.gca().add_artist(circle)
    plt.title(str(name)+" Class Distribution Level "+lvl)
    time = datetime.now().strftime("%H%M%S")
    plt.savefig('static/'+str(name)+lvl+time+'_class_dist_plot.png')
    plt.clf()
    return '/static/'+str(name)+lvl+time+'_class_dist_plot.png'
```

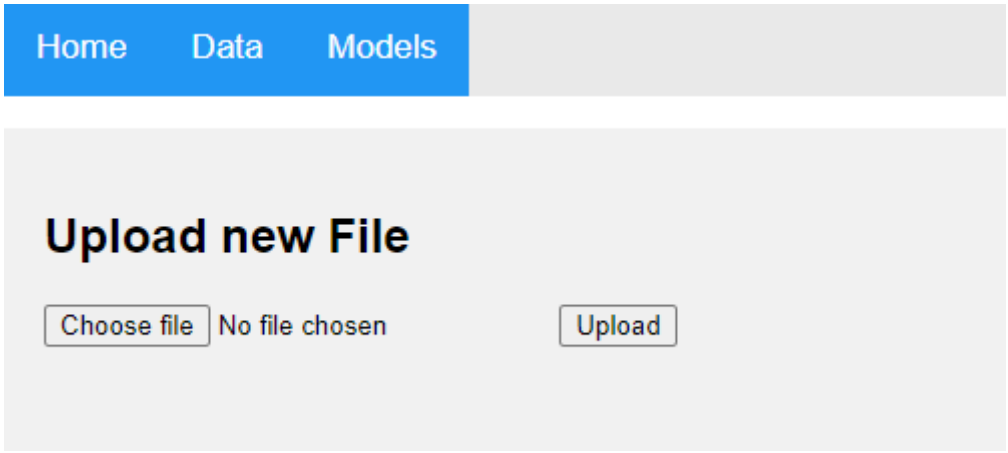
In [2]:

```
#Views metrics for a dataset
#@app.route('/data/metrics', methods=['POST'])
def metrics_data():
    c = request.form['dataMetricsName']
    try:
        if c not in ["test", "live"]: return render_template('data.html', outcome="F
        if c == "live":
            table = live_storage.fetch_all()
            class_dist = saveClassFig(table, "Live")
        else:
            return render_template('data.html', outcome="Form Error")
    except Exception:
        return render_template('data.html', outcome="Storage Error")
    print("Rendering")
    return render_template('data_metrics.html', label_figs=class_dist)
```

To plot the label distributions, we decided to use a pie chart. However, attempting to use a single pie chart to display all of the concatenated labels proved to be difficult to read. We therefore decided to divide the labels by hierarchy, giving 3 different pie charts corresponding to each level. The matplotlib module was used to plot the chart. This option is only available for test data, as the ground-truth labels are not recorded for live data entries.

Load CSV Data

The user can upload CSV file that contains test data. The file should have the correct headers as can be seen in the template file. Each text in the data must be classified with the classes according to the DBPedia Classes.



In [ ]:

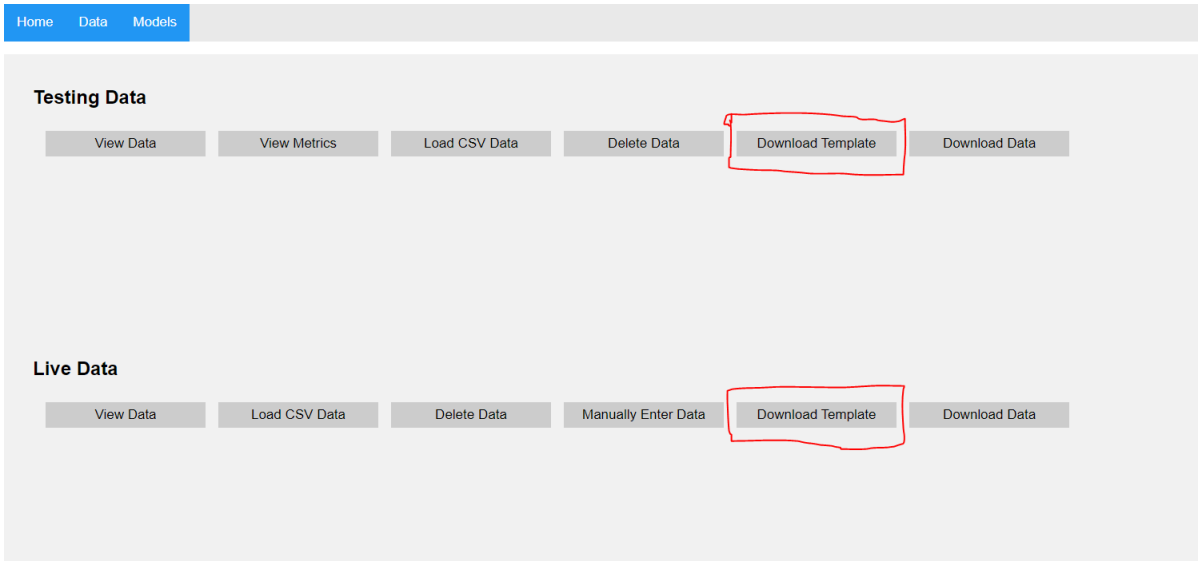
```
#Upload CSV Data
#@app.route('/data/load/upload', methods=['POST'])
def upload_data():
    try:
        if request.method == 'POST':
            if 'file' not in request.files:
                print('No file part')
                return redirect(request.url)
            file = request.files['file']
            if file.filename == '':
                print('No selected file')
                return redirect(request.url)
            if file and '.' in file.filename and file.filename.rsplit('.', 1)[1].low
                c = request.form['dataUploadName']
                if c == "test":
                    test_storage.load_file(file)
```

```
                table = test_storage.fetch_all()
            else:
                live_storage.load_file(file, isLive=True)
                table = live_storage.fetch_all()
            return render_template('upload.html', outcome="File Upload Successfu
    except Exception:
        return render_template('upload.html', outcome="File Upload Error")

    return render_template('upload.html')

#Loads data and then views it
#@app.route('/data/load', methods=['POST'])
def load_data():
    c = request.form['dataLoadName']
    if c not in ["test", "live"]: return render_template('data.html', outcome="Form
    print("Rendering")
    return render_template('upload.html', name=c)
```

To load data into the web application, we decided to take a primary approach. This approach involves loading in raw data from CSV files. To load them, from the data tab the user will select "load data". From here, they will be redirected to a file upload form where they may select a CSV data file from their local file system. Upon clicking "upload", the request will be checked to see if the file is A) a CSV file and is B) in the correct format. Should either of these prove false, the request will be rejected. If not, they will be processed and loaded into their respective storage device. The formatted data from storage will then be fetched, and the user will be redirected to the view data page to check what they have chosen to upload.

Download Template

To help the user understand what format the uploaded CSV files should take in terms of column headings, the user can download a template CSV file that gives column headings and example data entries.



In [ ]:

```
#Download CSV Template
#@app.route('/data/template', methods=['POST'])
def download_template():
    try:
        c = request.form['dataTemplateName']
        if c not in ["test", "live"]: return render_template('data.html', outcome="F
        if c == "test":
            return send_file('test_template.csv',
                     mimetype='text/csv',
```

```
                    attachment_filename='template.csv',
                    as_attachment=True)
        else:
            return send_file('live_template.csv',
                    mimetype='text/csv',
                    attachment_filename='template.csv',
                    as_attachment=True)
    except Exception:
        return render_template('data.html', outcome="Storage Error")
```

To facilitate this, a button is added to each storage type. When the user chooses to press these, the web application fetches the corresponding template file from the local server file system and downloads them for the user. They may then choose to use these to fill out additional CSV files for potential bulk updates in the test/live data. The contents for each of these templates are displayed here.

Test:

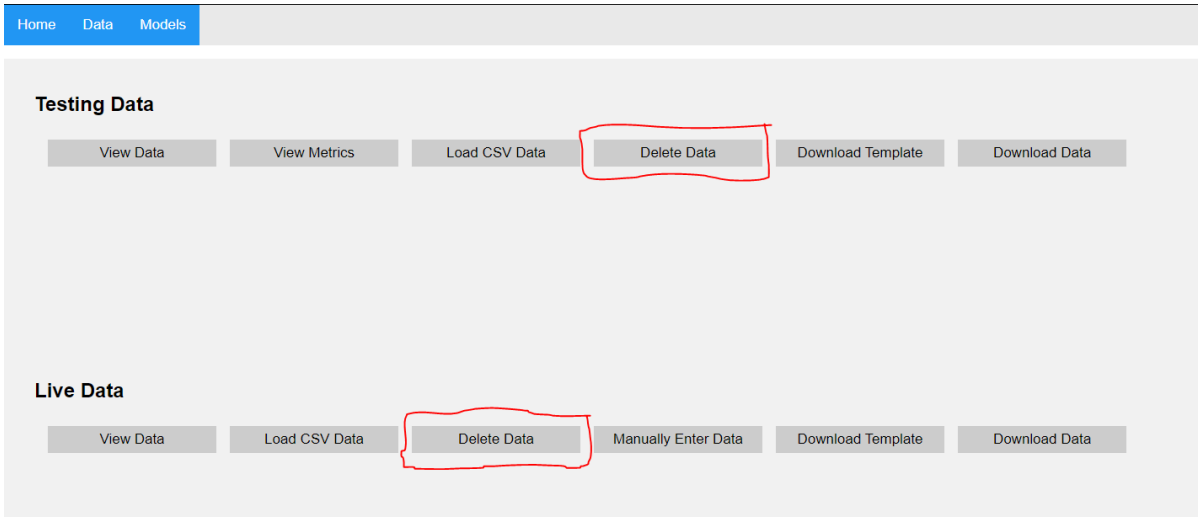
	A	B	C	D	E
1	text	l1	l2	l3	
2	Example	Agent	Politician	PrimeMinister	
3	...	...	...	...	
4					

Live:

	A	B	C	D
1	text			
2	Example			
3	...			

Delete Data

The user can delete the data that is currently being stored.



In [ ]:

```
#Deletes all data in storage x
#@app.route('/data/delete', methods=['POST'])
def delete_data():
    c = request.form['dataRemoveName']
    try:
        if c not in ["test", "live"]: return render_template('data.html', outcome="F
        if c == "test":
            test_storage.list = []
            test_storage.count = 0
        else:
            live_storage.list = []
```

```
            live_storage.count = 0
    except Exception:
        return render_template('data.html', outcome="Storage Error")
    print("Rendering")
    return render_template('data_view.html', table="Empty")
```

To delete data, a user may select the "Delete Data" button from the data tab. This will set the value of the respective storage list to empty, whilst resetting the storage counter.

Download Data

The user can choose to download the data currently held in storage.

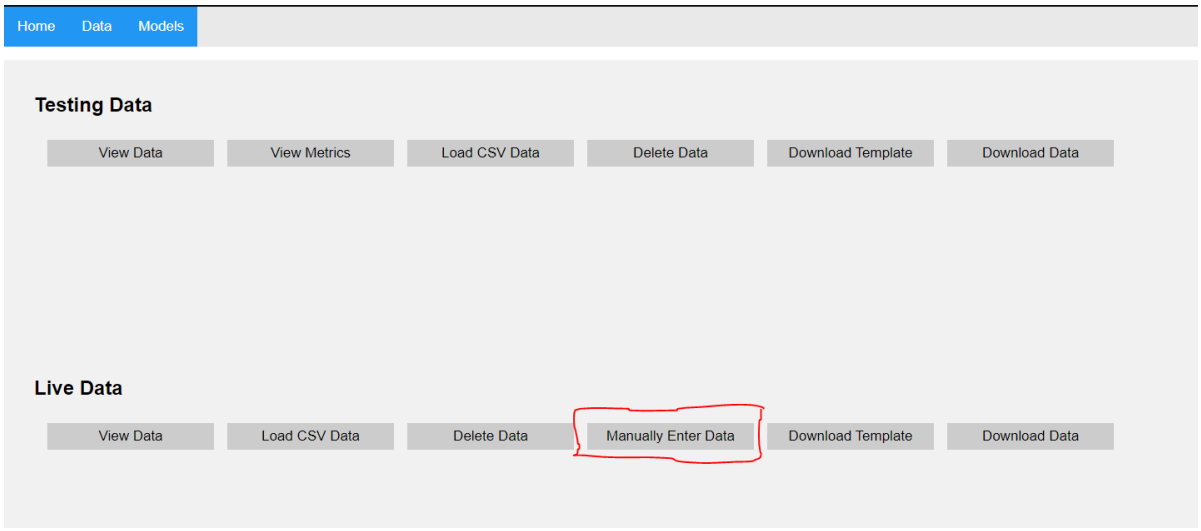
In [ ]:

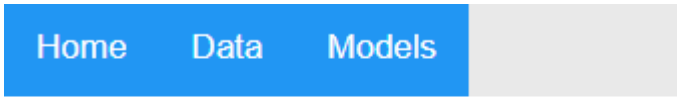
```
#Download Stored Data
#@app.route('/data/download', methods=['POST'])
def download_data():
    try:
        c = request.form['dataDownloadName']
        if c not in ["test", "live"]: return render_template('data.html', outcome="F
        if c == "test":
            table = test_storage.fetch_all()
            del table['label']
            csv = table.to_csv(index=False)
        else:
            table = live_storage.fetch_all()
            del table['label']
            csv = table.to_csv(index=False)
        return Response(
            csv,
            mimetype="text/csv",
            headers={"Content-disposition":
                     "attachment; filename=download.csv"})
    except Exception:
        return render_template('data.html', outcome="Storage Error")
```

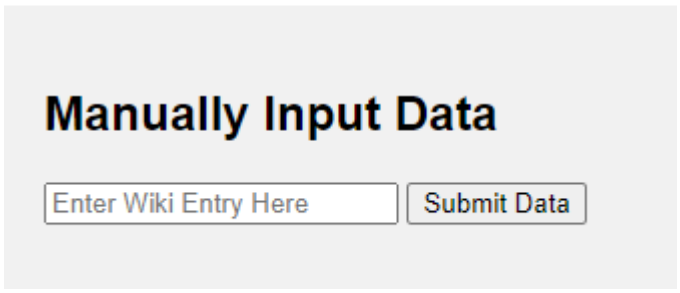
To download the stored data, a user may click the "Download Data" button on the data tab. The server will respond by fetching a dataframe of all the currently stored data from the selected storage object. It will then convert this dataframe to a CSV format and respond by downloading the CSV file for the client.

Manually Input Data

The user can add a single article using this function.







In [1]:

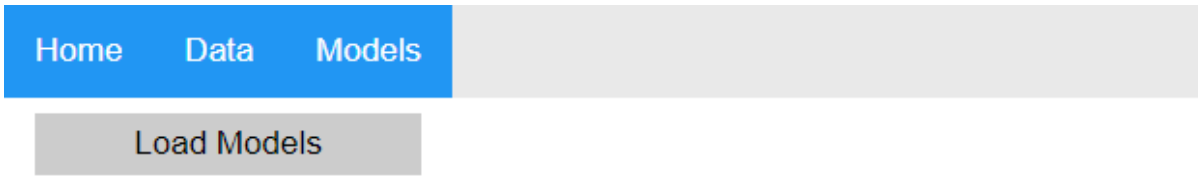
```
#Manually Enter Data
#@app.route('/data/load/manual', methods=['POST'])
def manual_data():
    c = request.form['dataManualName']
    if c not in ["live"]: return render_template('data.html', outcome="Form Error")
    print("Rendering")
    return render_template('data_manual.html', target=c, outcome="")

#Manually Enter Data Add
#@app.route('/data/load/manual/add', methods=['POST'])
def manual_data_add():
    print("manual add")
    c = request.form['type']
    text = request.form['text']
    try:
        if (c not in ["live"]) or (text == ""): return render_template('data_manual.
        live_storage.add(text, None, None)
    except Exception:
        return render_template('data_manual.html', outcome="Storage Error")
    print("Rendering")
    return render_template('data_manual.html', target=c, outcome="Data Entry Success
```

For live data, there is an option to manually enter data entries into the application storage via form. To do this, a user may click on the "Manually Enter Data" button on the data tab. From there, they will be redirected to a form where the article text may be entered and submitted.

Once submitted, the server will add a single entry to the live storage object. This feature is only included for the live data as an option to feed raw inputs into the model.

Models

This tab allows the user to load in all currently available models and pipelines as well use them to perform test evaluations or predictions.



The following is the class representation of the model and its pipeline. It encapsulates the two main functions attributed to it, being evaluation and prediction:

In [ ]:

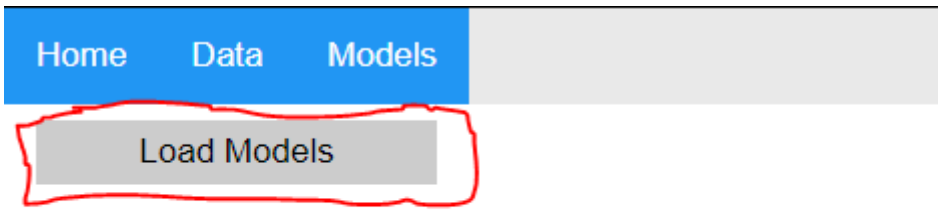
```
class Model():
    def __init__(self, pipeline, name):
        self.pipeline = pipeline
        self.name = name
        self.f1 = None
        self.roc_auc = None
        self.performance = None

        self.memory = []
        self.cpu = []
        self.time = []

    def evaluate(self, x, y_true):
        preds = self.pipeline.predict(x)
        y_preds = np.where(preds < 0.9, 0, 1)
        self.f1 = f1_score(y_true, y_preds, average='micro')*100
        self.roc_auc = roc_auc_score(y_true, preds, 'micro') * 100
        return y_preds

    def predict(self, x):
        preds = self.pipeline.predict(x)
        y_preds = np.where(preds < 0.9, 0, 1)
        return y_preds
```

Model Operations

The following is a list of the operations that can be performed with the models in this tab.

Load Models

This allows the user to load new models that have been added to the server file system.

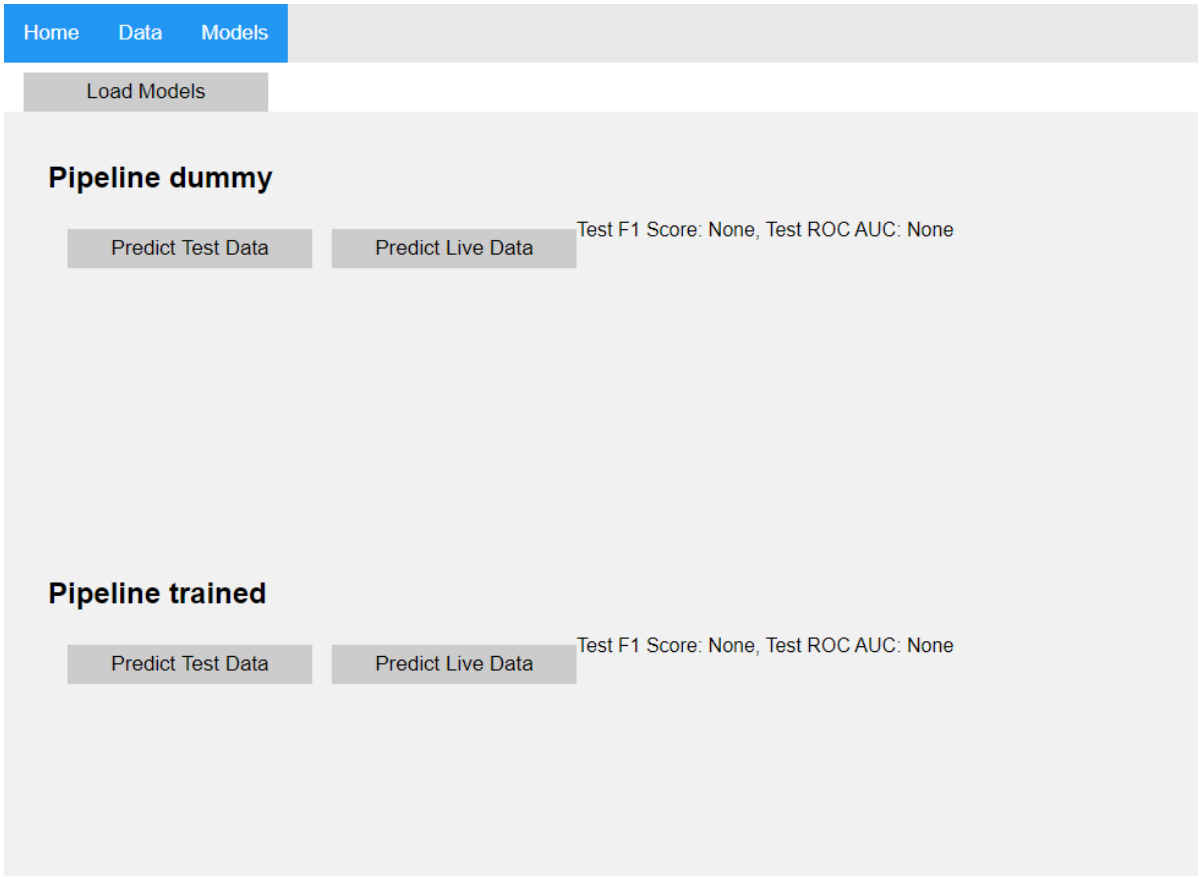


In [ ]:

```
#@app.before_first_request
def declare_vars():
    pd.set_option('display.min_rows', 65000)
    pd.set_option('display.max_rows', 65000)
    pd.set_option('display.max_columns', None)
    pd.set_option('display.width', None)
    pd.set_option('display.max_colwidth', 100)
    global pipelines
    global test_storage
    global live_storage
    pipelines = {}
    test_storage  = Storage()

    live_storage  = Storage()

#Load All Avaliable Models
#@app.route('/models/load', methods=['GET'])
def model_load():
    print("Loading Avaliable Models...")
    try:
        targets = [name for name in os.listdir(".") if (os.path.isdir(name) and "mod
        for target in targets:
            pipeline = load("pipeline_"+str(target.replace("model_", ""))+".joblib")
            pipeline.steps.insert(1,['classifier', load_model(target)])
            pipelines[target.replace("model_", "")] = Model(pipeline, target.replace
        print("Models Loaded!")
    except Exception:

        return render_template('models.html', models=pipelines, outcome="Pipeline Lo
    return render_template('models.html', models=pipelines)
```

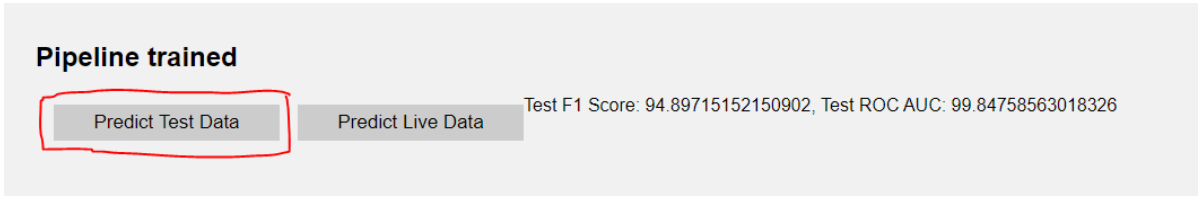
When the application begins, it declares the "pipelines" global variable. This is used to store a list of model objects which represents all of the currently loaded pipelines.

In the Models tab the user may choose to press the "Load Models" button, this triggers the load event which causes the server to search the local file system for directories beginning with "model_". If this is found, it will also find and load the corresponding pre-processing pipeline into the application. It will then create a new model object and store it in the pipelines global variable. This process is repeated for every model found in the file system. Once loaded, the models tab should look like the following (after loading two pipelines in this example):



Predict Test Data

This classifies the test data that have been uploaded and evaluates the model's performance using F1 score and ROC AUC score.



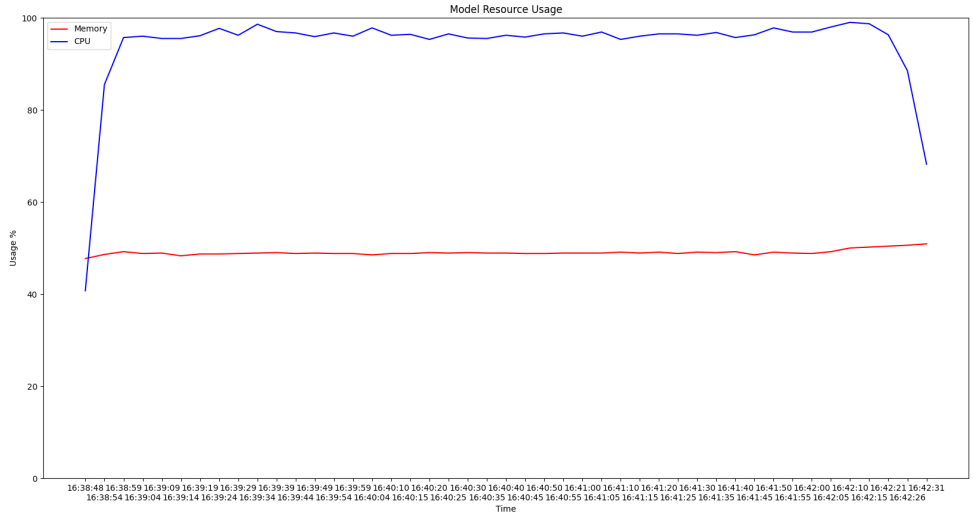
In [3]:

```
#Run Model Against Test Data
#@app.route('/models/run/test', methods=['POST'])
def model_test():
    try:
        test_data = test_storage.fetch_all()
        x_test = pd.DataFrame(data={'text': test_data['article_text']})
        y_test = test_data['label']
        y_test = np.array([a for a in y_test])
        thread = ResourceTracker(request.form['modelTestName'])
        thread.start()
        pred = pipelines[request.form['modelTestName']].evaluate(x_test, y_test)
        thread.stop()
        thread.join()
        print("Saving Predictions")
        for i in range(len(pred)):
            test_storage.list[i].prediction = pred[i]
        print("Predictions Saved")

        perf_fig = savePerformance(pipelines[request.form['modelTestName']].name, re
        pipelines[request.form['modelTestName']].performance = perf_fig
    except Exception:
        return render_template('models.html', models=pipelines, outcome="Model Error
```

```
    return render_template('models.html', models=pipelines)
```

To do this, when the user clicks "Predict Test Data" for any of the loaded models, it will first retrieve all current test data from storage. I will then feed the test data and labels into the evaluation method of the pipeline. This will set the objects f1 score and ROC AUC score in addition to returning a list of all of the predictions made.

In addition to this, before initiating the evaluation, the web application starts an additional background thread. This purpose is used for the purpose of taking regular measurements of the model's memory and cpu usage (assuming no available GPUs). Upon the conclusion of the prediction process, the graph showing these variations is displayed below the model options (Note the timestamps can be difficult to read after long executions).



In [ ]:

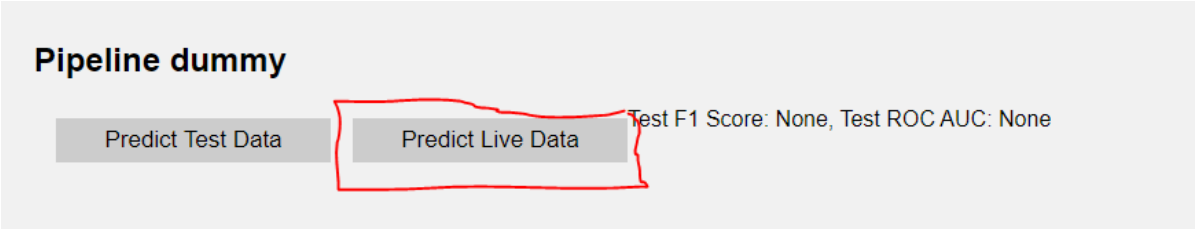
```
class ResourceTracker(threading.Thread):
    def __init__(self,  pipe_name):
        super(ResourceTracker, self).__init__()
        self._stop_event = threading.Event()
        self.flag = False
        self.pipe_name = pipe_name

    def run(self):
        while True:
            pipelines[self.pipe_name].memory.append(psutil.virtual_memory()[2])
            pipelines[self.pipe_name].cpu.append(psutil.cpu_percent(4))
            pipelines[self.pipe_name].time.append(datetime.now().strftime("%H:%M:%S"
            sleep(1)
            if self.flag:
                break
    def stop(self):
        self.flag = True
        self._stop_event.set()
```

This is achieved via the ResourceTracker class that overrides the threading.Thread python object. It uses the psutil module to take percentage measurements of system resources. The stop method is used to help the main thread communicate to this thread and to trigger a flag to stop it. Finally, the predictions made by the model can be viewed within the data view tab:

article_text	display_label	prediction
te on this line.	[Place, Station, RailwayStation]	[(Place, RailwayStation, Station)]
erated from 1...	[Place, Building, Hospital]	[(Building, Hospital, Place)]
as the Kurdis...	[Agent, Organisation, PoliticalParty]	[(Agent, Organisation, PoliticalParty)]
d based in, R...	[Agent, Person, Architect]	[(Agent, Architect, Person)]
the Romanian ...	[Agent, SportsTeam, HandballTeam]	[(Agent, HandballTeam, SportsTeam)]
sting radio, ...	[Agent, Broadcaster, BroadcastNetwork]	[(Agent, Broadcaster)]
, Switzerland...	[Place, NaturalPlace, Mountain]	[(Mountain, NaturalPlace, Place)]
at approximat...	[Device, Engine, AutomobileEngine]	[()]
otballer. He ...	[Agent, Athlete, SoccerPlayer]	[(Agent, Athlete, SoccerPlayer)]
7 years. It i...	[Place, CelestialBody, Planet]	[(CelestialBody, Place, Planet)]
singles, 14 v...	[Work, MusicalWork, ArtistDiscography]	[(ArtistDiscography, MusicalWork, Work)]
became a Hus...	[Agent, Person, Noble]	[(Agent, Noble, Person)]

Predict Live Data

This classifies the live data that have been uploaded. This method simply records the predictions (as shown above), but does not include any performance metrics like F1 score as the model is not being evaluated in this case.



In [ ]:

```
#Run Model Against Live Data
#@app.route('/models/run/live', methods=['POST'])
def model_live():
    try:
        live_data = live_storage.fetch_all()
        x_test = pd.DataFrame(data={'text': live_data['article_text']})
        thread = ResourceTracker(request.form['modelLiveName'])
        thread.start()
        pred = pipelines[request.form['modelLiveName']].predict(x_test)
        thread.stop()
        thread.join()
        print(pipelines[request.form['modelLiveName']].time, pipelines[request.form[

        print("Saving Predictions")
        for i in range(len(pred)):
           live_storage.list[i].prediction = pred[i]
        print("Predictions Saved")

        perf_fig = savePerformance(pipelines[request.form['modelLiveName']].name, re
        pipelines[request.form['modelLiveName']].performance = perf_fig
    except Exception:
        return render_template('models.html', models=pipelines, outcome="Model Error

    return render_template('models.html', models=pipelines)
```

The server uses an almost identical response to the live prediction request as the test one. However, in this case the "prediction()" method for the Model object is called instead of "evaluate()".

Task 3: Test Results

We have manually tested the web application via a browser and added code to handle exceptions.

After that, unit testing (unit testing.py) has been done to check the web service performs as expected. There are 15 tests to check the following actions:

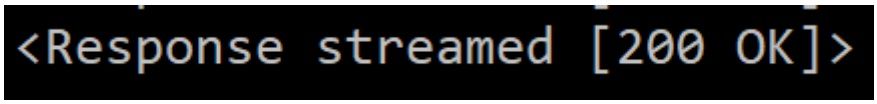
- Open Home tab
- Open Data tab
- Open Models tab
- Open data upload page
- Upload CSV file
- View data
- View metrics
- Open live data manual input page
- Input live data manually
- Download a template
- Load models
- Predict test data
- Predict live data
- Delete data
- Download data

Example of a unit test:

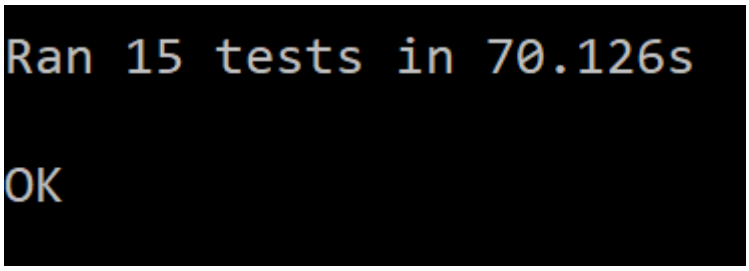
In [4]:
```
def test_get_data(self):
        tester = app.test_client(self)
        response = tester.get('/data')
        print(response)
        self.assertEqual(response.status_code, 200)
```

All of these tests have been performed using the python unittest module.

Every test returned the following response.



The final result can be seen below.



From the result, it can be said that the web service performs as expected.

Task 4: Performance

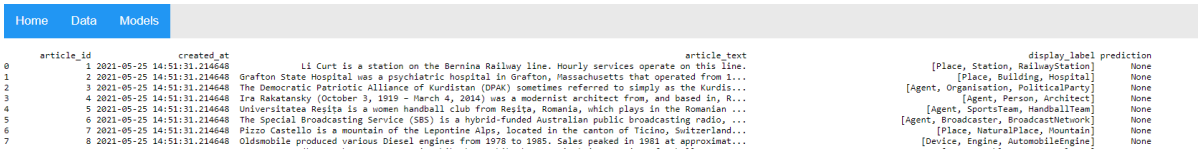
We have successfully developed a web service that allows a user to classify articles into multiple classes. The web service allows not only classify articles but also test the articles that have already been classified. This lets the user to check if the articles have been classified correctly. Data can be viewed, deleted, and downloaded in CSV format. The user can also check the test results with both F1 score and ROC AUC score. Templates for uploading data can be

downloaded via the web service which makes it easier for the user to create their own data file. After uploading test data, the user can view the class distributions for each level of classes. The web service let the user to add live data in two ways: manual input and file upload. Enabling the manual input lets the user to get the classes of an article easily. All versions of the models that are available on the web service can be loaded and the user can select any model for predictions. The web service displays the model resource usage including CPU usage and memory usage for a prediction with a line graph. It handles exceptions that may disrupt the flow of execution in the web service.

Although there are many useful features as mentioned above, there are some downsides. One of the downsides of the web service is that it takes time to load models and classify a lot of articles. Another downside is the test results that have been provided previously get removed when loading models. Also, there is a limitation for displaying data that it cannot display too many entries when viewing the data. Displaying this in multiple pages would improve the performance.

Task 5: Monitoring

As mentioned in Task 2, the user inputs and the model predictions are stored and can be viewed on the "View Data" page under the Data tab:



This data can also be downloaded for further use in the Data tab:

In addition to monitoring of data and models, the web service also monitors the memory & CPU usage for predictions via threading. This is also mentioned in Task 2.

Task 6: CI/CD Pipeline

When adding a new model to the web server, a new pipeline must be created. Following code creates a pipeline for the new model.

In [27]:
```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
```

In [28]:
```
class Preprocessor(BaseEstimator, TransformerMixin):
    def __init__(self, x):
        self.stops = set(stopwords.words("english"))
        self.maxlen = 500
        self.num_words = 250000
        x['text']  = x.text.map(lambda x: self.remove_punctuation(x))
        x['text']  = x['text'].map(self.remove_stopwords)
        self.tokenizer = Tokenizer(num_words=self.num_words)
        self.tokenizer.fit_on_texts(x.text)

    def fit(self, x, y=None):
        x['text'] = x.text.map(lambda x: self.remove_punctuation(x))
        x['text'] = x['text'].map(self.remove_stopwords)
        x = self.to_sequences(x.text)
        return x

    def transform(self, x):
        x['text'] = x.text.map(lambda x: self.remove_punctuation(x))
        x['text'] = x['text'].map(self.remove_stopwords)
        x = self.to_sequences(x.text)
        return x

    def fit_transform(self, x, y):
        x = x.copy()
        x['text'] = x.text.map(lambda x: self.remove_punctuation(x))
        x['text'] = x['text'].map(self.remove_stopwords)
        x = self.to_sequences(x.text)
        return x

    def remove_punctuation(self, text):
        table = str.maketrans("","", string.punctuation)
        return text.translate(table)

    def remove_stopwords(self, text):
        text = [word.lower() for word in text.split() if word.lower() not in self.st
        return " ".join(text)

    def to_sequences(self, tokens):
        seq = self.tokenizer.texts_to_sequences(tokens)
        seq = pad_sequences(seq, maxlen=self.maxlen, padding="post", truncating="pos
        return seq
```

In [30]:
```
train = pd.read_csv('./data/DBPEDIA_train.csv')
```

In [31]:
```
pipeline = Pipeline(steps=[('preprocess', Preprocessor(train))])
```

In [32]:
```
filename = "pipeline_"+name+".joblib"
dump(pipeline, filename=filename)
```

Out[32]:
```
['pipeline_first.joblib']
```

Name of the pipeline must be the same as the name of the model. For example if the model is named "model_version1", the name of the pipeline must be "pipeline_version1".

After adding the new model and pipeline to the web server, these can be loaded by clicking the "Load Models" button under the Model tab and used for classification.

In [ ]: