CS 475/675 Machine Learning: Homework 4
Supervised Classifiers 2
Programming Assignment

Due: Thursday, October 29, 2020, 11:59 pm
40 Points Total Version 1.0

Make sure to read from start to finish before beginning the assignment.

This assignment must be completed individually. No partners allowed.

1 Introduction

This assignment is all about SVMs. From lecture, you know that SVMs present an optimization problem with both primal and dual formulations. You will be implementing a kernel SVM that uses kernels suitable for text data. Your kernel SVM will be used for binary text classification: identify the topic of a news articles.

We have provided Python code to serve as a testbed for your algorithms. We'll use the same coding framework that we used in homework 2. You will fill in the details. Search for comments that begin with `TODO`; these sections need to be written. Your code for this assignment will all be within the `models.py` and `kernels.py` files; do not modify any of the other files.

You will be implementing one kernel SVM algorithm and two kernel functions. We recommend you implement the n-gram kernel first, followed by the kernel SVM, and finally by the tf-idf word kernel.

2 Data

We will be using documents from the `Reuters-21578` dataset[1]. The documents in this dataset contain stories that appeared on the Reuters newswire in 1987. Train and test sets were extracted from the corpus using the Modified Apte split, resulting in 9,603 and 3,299 articles in each split, respectively. Both splits were then processed using standard techniques from NLP, including tokenization, converting all words to lowercase, and removing stopwords[2], punctuation, and numbers. We have provided you with all of the processing code to extract text classification tasks from the raw SGML[3] files in `data.py`. While you don't have to understand what this script is doing for the purposes of this assignment, you may find it helpful for future ML/NLP projects.

[1] `https://archive.ics.uci.edu/ml/datasets/reuters-21578+text+categorization+collection`
[2] `https://en.wikipedia.org/wiki/Stop_word`
[3] `https://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language`

The Reuters Corpus is a text categorization test collection, which means that each document was hand-labeled with pre-specified categories. There are 118 different categories within the corpus, which means 118 binary text classification tasks can be extracted. However, only 10 of these categories have a substantial number of train/test articles. You will want to use categories from these 10 options (train, test article count listed with the category name):

- earn (2877, 1087)
- acquisitions (1650, 179)
- money-fx (538, 179)
- grain (433, 149)
- crude (389, 189)
- trade (369, 119)
- interest (347, 131)
- ship (197, 89)
- wheat (212, 71)
- corn (182, 56)

See section 3.1 for details on how to use these categories to define a binary classification task for your kernel SVM.

3 Existing Components

The foundations of the learning framework have been provided for you. You will need to complete this library by filling in code where you see a `TODO` comment. **You should only make changes to `models.py` and `kernels.py`. Do not change the existing command line flags, existing filenames, or algorithm names.** We use these command lines to test your code. If you change their behavior, we cannot test your code.

The code we have provided is fairly compact, and you should spend some time to familiarize yourself with it. Here is a short summary to get you started:

- `data.py` – This contains the `load_data` function, which extracts text classification tasks from the Reuters Corpus and returns a list of documents and labels. Here, a document is a single string with all of the words and whitespace concatenated. The labels are stored as a list of category strings. In `classify.py`, this is binarized based on the category you are predicting.
- `classify.py` – This is the main testbed to be run from the command line. It takes care of parsing command line arguments, entering train/test mode, saving models/predictions, etc. You should not need to make any changes in this file this time.

- `models.py` – This contains a `Model` class which you should extend. Models have (in the very least) a `fit` method, for fitting the model to data, and a `predict` method, which computes predictions from features. You are free to add other methods as necessary. **Note that all predictions from your model must be 0 or 1; if you use other intermediate values for internal computations, then they must be converted before they are returned.**

- `kernels.py` – This contains a `Kernel` class which you should extend. Kernels have (in the very least) an `evaluate` method, for evaluating the kernel between two examples, and a `compute_kernel_matrix` method, which precomputes a kernel matrix which you will index into in your SVM implementation. You are free to add other methods as necessary. Note that the `compute_kernel_matrix` has a `TODO` in the base class. This is because populating the kernel matrix should work the same regardless of which kernel function you are using. **You should not change how `evaluate` and `compute_kernel_matrix` are called**. However, you are free to use or not use any other helper functions defined (i.e. the helper functions for computing the tf-idf in `TFIDFKernel`).

- `compute_accuracy.py` – This is a script which simply compares the true labels from a binary classification task (e.g., predict `earn` given `earn acq`) to the predictions that were saved by running `classify.py` (e.g., `ngram.earn.predictions`).

- `run.sh` – For your convenience, we have included a bash script that defines the `train`, `test`, and `compute_accuracy` commands (no more needing to write all those command arguments!). If you open the bash script in your text editor, you can specify the data directory, topics, and kernel function that are passed to the respective commands. You can run the script as follows:

```
bash run.sh
```

3.1 How to Run the Provided Framework

The framework operates in two modes: train and test. Both stages are defined in `classify.py`.

3.1.1 Train Mode

The usage for train mode is

```
python3 classify.py --mode train --topics first_topic second_topic --kernel kernel_function \
                    --model-file model_file --datadir data_directory
```

The `mode` option indicates which mode to run (train or test). The `kernel` option indicates which kernel function to use (ngram or tfidf). Each assignment will specify the string argument for an algorithm. The `datadir` option indicates the path to the Reuters SGML files. The `topics` option takes two arguments, which correspond to the topics of articles to extract from the Reuters corpus. The first topic sets the binary classification task (i.e. that is the topic your SVM will be predicting). Finally, the `model-file` option specifies where to save the trained model.

3.1.2 Test Mode

The test mode is run in a similar manner:

```
python3 classify.py --mode test --topics first_topic second_topic --kernel kernel_function \
                    --model-file model_file --datadir data_directory --predictions-file predictions_file
```

The `model_file` is loaded and run on the `data`. Results are saved to the `predictions_file`.

3.1.3 Examples

As an example, the following trains the SVM using the ngram kernel function to predict the topic "earn":

```
python3 classify.py --mode train --kernel ngram --topics earn acq --model-file ngram.earn.model \
                     --datadir datasets/
```

To run the trained model on development data:

```
python3 classify.py --mode test --kernel ngram --topics earn acq --model-file ngram.earn.model \
                     --datadir datasets/ --predictions-file ngram.earn.predictions
```

Model predictions are saved as one predicted label per line in the same order as the input data. The code that generates these predictions is provided in the library. The script `compute_accuracy.py` can be used to evaluate the accuracy of your predictions for classification:

```
python3 compute_accuracy.py --datadir data_directory --topics first_topic second_topic \
                            --predictions-file predictions_file
```

We provide this script since it is exactly how we will evaluate your output. Make sure that your algorithm is outputting labels as they appear in the input files. If you use a different internal representation of your labels, make sure the output matches what's in the data files. The above script will do this for you, as you'll get low accuracy if you write the wrong labels.

4 Kernel Support Vector Machines (25 points)

A Support Vector Machine constructs a hyperplane in high dimensional space that separates training points of different classes while keeping a large margin with regards to the training points closest to the hyperplane. Standard SVMs are linear, binary classifiers. But not all data is linearly separable. We can use kernels to project our data into a higher dimensional space and obtain non-linear decision boundaries. Consider a positive-definite real-valued kernel function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Then, for all $x, x' \in \mathcal{X}$, the kernel function $K(x, x')$ can be expressed as the inner product in another space $\mathcal{V}$. This stems from the Representer Theorem[4], which implies that the optimal solution to the primal SVM objective function can be expressed as a linear combination of the training instances.

This makes it possible to train and use the SVM *without* direct access to the training instances, and instead with *only* access to their inner products specified by the kernel operator $K(x, x')$. Then, instead of considering predictors which are linear functions of the training instances $\mathbf{X}$ themselves, we consider predictors which are linear functions of some implicit mapping $\phi : \mathcal{X} \to \mathcal{V}$ of the training data. SVMs are typically formulated as

[4]Schölkopf, Bernhard; Herbrich, Ralf; Smola, Alex J. (2001). A Generalized Representer Theorem. Computational Learning Theory. Lecture Notes in Computer Science. 2111. pp. 416–426.

a constrained quadratic programming problem. We can also reformulate it as an equivalent unconstrained problem of empirical loss minimization plus a regularization term. Formally, given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^M$ and $y_i \in \{+1, -1\}$, we want to find the minimizer of the problem:

$$\mathbf{w}^\star = \underset{\mathbf{w}}{\operatorname{argmin}} \left[\lambda \frac{1}{2} \|\mathbf{w}\|^2 + \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{w}; (\phi(\mathbf{x}_i), y_i)) \right] \tag{1}$$

where λ is the regularization parameter,

$$l(\mathbf{w}; (\phi(\mathbf{x}), y)) = \max\{0, 1 - y\langle \mathbf{w}, \phi(\mathbf{x}) \rangle\} \tag{2}$$

and $\langle \cdot, \cdot \rangle$ is the inner product operator.

As you might have noticed, we omit the parameter for the bias term throughout this homework (i.e., the hyperplane is always across the origin).

To solve the optimization problem defined in Eq. 1, typically we switch to the dual formulation. To do so, we rewrite the primal problem as a function of α variables and then take gradients with respect to α. The Representer Theorem guarantees that the optimal solution to Eq. 1 is spanned by the training instances and is of the form $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \phi(\mathbf{x}_i)$. The training objective can then be written in terms of α variables and kernel evaluations:

$$\alpha^\star = \underset{\alpha}{\operatorname{argmin}} \left[\frac{\lambda}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) + \frac{1}{N} \sum_{i=1}^{N} \max \left\{ 0, 1 - y_i \sum_{j=1}^{N} \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) \right\} \right] \tag{3}$$

With this formation, the feature mapping $\phi(\cdot)$ is never explicitly specified, but instead defined through the kernel operator $K(x, x') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$. In this programming assignment, we will rewrite the loss function in terms of α to take advantage of the kernel operator $K(x, x')$ **but we will compute the gradient with respect to w** so technically we are still solving the **primal** problem. *Equation 3 is* not *the approach we will take in this assignment.*

4.1 Kernel Pegasos

Pegasos is a simple but effective stochastic gradient descent algorithm to solve a Support Vector Machine for binary classification problems. The approach is called *Primal Estimated sub-GrAdient SOlver for SVM* (Pegasos).[5] The number of iterations required by Pegasos to obtain a solution of accuracy ϵ is $O(1/\epsilon)$, while previous stochastic gradient descent methods require $O(1/\epsilon^2)$. You will implement an extension of the Pegasos algorithm that uses kernels while still minimizing the primal problem. The number of iterations required by Kernel Pegasos to obtain a solution of accuracy ϵ is $O(1/(\lambda\epsilon))$[6].

The kernel version of Pagasos uses only kernel evaluations and doesn't explicitly access

[5] Shalev-Shwartz, S., Singer, Y., Srebro, N., Cotter, A. (2011). Pegasos: Primal estimated sub-gradient solver for svm. Mathematical programming, 127(1), 3-30.

[6] The number of iterations required does not depend on the number of training examples, however your runtime does. This is because we are checking for non-zero loss at each iteration t, so we may require as much as $\min(t, N)$ kernel evaluations. This brings the overall runtime to $O(N/(\lambda\epsilon))$.

feature vectors $\phi(\mathbf{x})$ or weight vector $\mathbf{w}$. The key to performing the kernel trick is to substitute dual variables into the objective and to keep track of a variable α_i for each data point. Our α_i's will then indirectly update our weight matrix for us:

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \phi(\mathbf{x}_i) \tag{4}$$

4.2 Kernel Trick[7]

So what are these α_i values? Let's derive the update rules for α_i and then see how these relate to our weight updates. We can replace the objective in Eq. 1 with an approximate based indirectly on $(\phi(\mathbf{x})^{(t)}, y^{(t)})$, yielding:

$$f(\mathbf{w}; i^{(t)}) = \lambda \frac{1}{2} \|\mathbf{w}\|^2 + l(\mathbf{w}; (\phi(\mathbf{x}^{(t)}), y^{(t)})) \tag{5}$$

We consider the sub-gradient of the above approximate objective, given by:

$$\frac{\partial f(\mathbf{w}^{(t)}; i^{(t)})}{\partial \mathbf{w}^{(t)}} = \lambda \mathbf{w}^{(t)} - \mathbb{1}[y^{(t)} \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)}) \rangle < 1] y^{(t)} \phi(\mathbf{x}^{(t)}) \tag{6}$$

where $\mathbb{1}[\cdot]$ is the indicator function which takes a value of 1 if its argument is true, and 0 otherwise. We then update $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \frac{\partial f(\mathbf{w}^{(t)}; i^{(t)})}{\partial \mathbf{w}^{(t)}}$ using a step size of $\eta^{(t)} = 1/(\lambda t)$. Note that this update can be written as:

$$\mathbf{w}^{(t+1)} \leftarrow \left(1 - \frac{1}{t}\right) \mathbf{w}^{(t)} + \frac{1}{\lambda t} \mathbb{1}[y^{(t)} \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)}) \rangle < 1] y^{(t)} \phi(\mathbf{x}^{(t)}) \tag{7}$$

Multiplying both sides by t and then rearranging, we get:

$$t\mathbf{w}^{(t+1)} - (t-1)\mathbf{w}^{(t)} = \frac{1}{\lambda} \mathbb{1}[y^{(t)} \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)}) \rangle < 1] y^{(t)} \phi(\mathbf{x}^{(t)}) \tag{8}$$

Observe that Eq. 8 holds for any value t, which gives us a system of t equations:

$$\begin{cases} t\mathbf{w}^{(t+1)} - (t-1)\mathbf{w}^{(t)} & = \frac{1}{\lambda} \mathbb{1}[y^{(t)} \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)}) \rangle < 1] y^{(t)} \phi(\mathbf{x}^{(t)}) \\ (t-1)\mathbf{w}^{(t)} - (t-2)\mathbf{w}^{(t-1)} & = \frac{1}{\lambda} \mathbb{1}[y^{(t-1)} \langle \mathbf{w}^{(t-1)}, \phi(\mathbf{x}^{(t-1)}) \rangle < 1] y^{(t-1)} \phi(\mathbf{x}^{(t-1)}) \\ & \dots \\ \mathbf{w}^{(2)} & = \frac{1}{\lambda} \mathbb{1}[y^{(1)} \langle \mathbf{w}^{(1)}, \phi(\mathbf{x}^{(1)}) \rangle < 1] y^{(1)} \phi(\mathbf{x}^{(1)}) \end{cases}$$

Now, by summing over these t equations and then dividing both sides by t, we get

$$\mathbf{w}^{(t+1)} = \frac{1}{\lambda t} \sum_{k=1}^{t} \mathbb{1}[y^{(k)} \langle \mathbf{w}^{(k)}, \phi(\mathbf{x}^{(k)}) \rangle < 1] y^{(k)} \phi(\mathbf{x}^{(k)}) \tag{9}$$

[7]Typically, the dual formulation is used for kernel SVMs because the feature mapping can be specified in terms of a kernel function. However, the Pegasos algorithm allows us to minimize the primal objective in terms of kernel operators. As you'll see, we will substitute in the dual formulas to reap this benefit while still solving the primal problem.

To match our desired form in Eq. 4, we can rewrite this in terms of a summation over i:

$$\mathbf{w}^{(t+1)} = \sum_i \underbrace{\left(\frac{1}{\lambda t} \sum_{k=1}^{t} \mathbb{1}[(\phi(\mathbf{x}_i), y_i) = (\phi(\mathbf{x}^{(k)}), y^{(k)})] \cdot \mathbb{1}[y^{(k)} \langle \mathbf{w}^{(k)}, \phi(\mathbf{x}^{(k)}) \rangle < 1] \right)}_{\alpha_i^{(t+1)}} y_i \phi(\mathbf{x}_i) \tag{10}$$

Then, for every t, $\lambda t \alpha_i^{(t+1)}$ counts how many times example i appears before iteration t (the indicator function on the left) and also satisfies $y_i \langle \mathbf{w}^{(k)}, \phi(\mathbf{x}_i) \rangle < 1$ (the indicator function on the right). This implies a simple update rule for α_i:

$$\lambda t \alpha_i^{(t+1)} \leftarrow \lambda(t-1)\alpha_i^{(t)} + \mathbb{1}[(\phi(\mathbf{x}_i), y_i) = (\phi(\mathbf{x}^{(t)}), y^{(t)})] \cdot \mathbb{1}[y_i \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}_i) \rangle < 1] \tag{11}$$

Suppose we draw data point $(\mathbf{x}_i, y_i)$ at iteration t. Then $\lambda t \alpha_i$ is incremented by 1 if and only if $y_i \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}_i) \rangle < 1$, i.e., if the prediction is incorrect.

4.3 Training

Kernel Pegasos performs stochastic gradient descent on the objective in Eq. 1. The learning rate decreases with each iteration to guarantee convergence. For further details, see Section 4.5.

Many descriptions of SGD call for shuffling your data before learning. This is a good idea to break any dependence in the order of your data. In order to achieve consistent results for everyone, we are requiring that you **do not shuffle your data**. When you are training, you will go through your data in the order it appeared in the data file.

Kernel Pegasos operates as follows. We initialize $\alpha_0 = 0$ at the start of the first iteration. We then make one slight modification to the training algorithm. Typically, at each time step t of the algorithm (starting from $t = 1$), we would choose a random training example $(\mathbf{x}^{(t)}, y^{(t)})$ by picking an index $i^{(t)} \in \{1, \ldots, N\}$ uniformly at random. **For your implementation, you should not select the training example randomly, and instead iterate through the examples in order.** For further details, see Section 4.6.

For simplicity, let's denote $\beta^{(t)} = \lambda(t-1)\alpha^{(t)}$.[8] Then, we can simplify the update rule defined in Eq. 11 to:

$$\beta_i^{(t+1)} \leftarrow \beta_i^{(t)} + \mathbb{1}[(\phi(\mathbf{x}_i), y_i) = (\phi(\mathbf{x}^{(t)}), y^{(t)})] \cdot \mathbb{1}[y_k \langle \mathbf{w}^{(t)}, \phi(\mathbf{x}^{(t)}) \rangle < 1] \tag{12}$$

Recall Eq. 4. Instead of keeping in memory the weight vector $\mathbf{w}^{(t+1)}$, we will represent $\mathbf{w}^{(t+1)}$ using $\beta^{(t+1)}$ by:

$$\mathbf{w}^{(t+1)} = \frac{1}{\lambda t} \sum_{i=1}^{N} \beta_i^{(t+1)} y_i \phi(\mathbf{x}_i) \tag{13}$$

[8] We recommend coding your implementation in terms of β rather than α for simplicity.

By equations 12 and 13, this update equation tells us that when we draw data point $(\mathbf{x}_j, y_j)$ at time step t, then we increment β_j by 1 (i.e., $\beta_j^{(t+1)} \leftarrow \beta_j^{(t)} + 1$) if and only if:

$$\frac{y_j}{\lambda(t-1)} \sum_{i=1}^{N} \beta_i^{(t)} y_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle < 1 \tag{14}$$

Note that only one element of β is changed at each time step. Additionally, notice that we haven't defined explicitly what $\phi(\cdot)$ is. This is because your algorithm should refer only to the kernel evaluations and not an explicit feature mapping, i.e., we can rewrite Eq. 14:

$$\frac{y_j}{\lambda(t-1)} \sum_{i=1}^{N} \beta_i^{(t)} y_i K(\mathbf{x}_i, \mathbf{x}_j) < 1 \tag{15}$$

Since we initialize $t = 1$, to avoid divide by 0 errors, you should increment t before performing the update. We finish training after a predetermined total number of time steps T, which equals the total number of examples times the total number of epochs.

To be clear, a single time step t should be a single example. You should update β after seeing a single example, not in batch mode.

4.4 Prediction

We can recover our α_i values from β_i by:

$$\alpha_i = \frac{1}{\lambda T} \beta_i^{(T+1)} \tag{16}$$

where T is the total number of time steps (the product of the number of training examples and the number of epochs) taken during training. Then, during prediction, we use the following formula:

$$\hat{y} = \text{sign}\left(\langle \mathbf{w}, \mathbf{x} \rangle\right) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right) \tag{17}$$

Notice above that you'll have to recompute a kernel matrix using your support vectors and the test data. **You should save your support vectors when fitting your SVM**.

Note that you will need to map the $\hat{y}$ values of ± 1 back to $\{0, 1\}$ for the `predict` function to return.

4.5 Learning Rate

We adopt a simple strategy to decreasing the learning rate: we make the learning rate a function of the time steps. Specifically, your learning rate should be $\eta^{(t)} = 1/(\lambda t)$. t is initialized to 1, incremented with each data point, and is **not** reset with each iteration through the data.

4.6 Sampling Instances

Pegasos relies on sampling examples for each time step. **For simplicity, we will NOT randomly sample instances. Instead, on round t you should update using the tth example** (modulo the size of the data set). An epoch involves a single pass in order through all the provided instances. You will make several passes through the data based on `--train-epochs`.

4.7 Regularization Parameter

Pegasos uses a regularization parameter λ to adjust the relative strength of regularization. The default value for λ is 10^{-4}. This parameter can be adjusted via the command line argument `--pegasos-lambda`.

4.8 Offset Feature

None of the math above mentions an offset feature (bias feature) $\mathbf{w}_0$, that corresponds to a $x_{i,0}$ that is always 1. It turns out that we don't need this if our data is centered. By centered we mean that $E[y] = 0$. For simplicity, assume that the data used in this assignment is centered (even though this may not be true). Do not include another feature that is always 1 (x_0) or weight ($\mathbf{w}_0$) for it.

4.9 Convergence

In practice, SGD optimization requires the program to determine when it has converged. Ideally, a maximized function has a gradient value of 0, but due to issues related to your step size, random noise, and machine precision, your gradient will likely never be exactly zero. Common practice is to check that the L_p norm of the gradient is less than some δ, for some p. For the sake of simplicity and consistent results, we will not do this in this assignment. Instead, your program should take a parameter `--train-epochs` which is *exactly* how many epochs you should run (not an upper bound). An epoch is a single pass over every training example. **Note that the "t" in Section 4 indexes a time step, and each epoch includes a time step for each training example.** The default of `--train-epochs` is 5.

5 Kernel Function (15 points)

During your updates, you will obtain the feature maps $\phi(\cdot)$ by evaluating kernels suitable for text data. In particular, you will implement a linear kernel that measures the similarity between documents indexed by n-grams (`NgramKernel`) and another linear kernel that measures the similarity between documents indexed by tfidf-weighted words (`TFIDFKernel`).

5.1 n-gram Kernel

An character n-gram is a contiguous sequence of n characters in a string. Since our documents consist of words and whitespace concatenated together, the n-grams can contain spaces (and dashes in the case of hyphenated words). Let's look at an example. Suppose we set $n = 3$, then from the phrase "`the black cat`", we can obtain the following 11 trigrams: "the", "he ", "e b", " bl", "bla", "lac", "ack", "ck ", "k c", " ca", "cat".

Given two documents x and x', we first generate the set of n-grams of length n. Then, the kernel can be computed as follows:

$$K(x, x') = \begin{cases} 1 & \text{if } x \cup x' = \emptyset \\ |x \cap x'| \,/\, |x \cup x'| & \text{otherwise} \end{cases} \tag{18}$$

You can specify the n-gram length using the command line argument `--ngram-length`. The default value is 3. You may find it useful to use Python's set arithmetic for an efficient implementation.

5.2 tf-idf Kernel

tf-idf (term frequency-inverse document frequency) is a statistic that quantifies the importance of a word to a document within a collection of documents[9]. Variations of tf-idf are often used in information-retrieval applications like document classification and scoring and ranking a document's relevence to a query in search engines. The building blocks of tf-idf include:

- Term Frequency: The count of occurrences a word in a particular document.

$$\text{tf}(t, d) = \text{count of word } w \text{ in document } d \,/\, \text{number of words in } d \tag{19}$$

- Document Frequency: The number of documents a particular word appears in. This measures the importance of a document within the collection of documents.

$$\text{df}(w) = \text{count of documents word } w \text{ appears in} \tag{20}$$

- Inverse Document Frequency: One issue with using the term frequency as a measure of the importance of a word in the corpus is that certain words like "the" and "this" will occur with high frequency in many documents, which may overpower less frequent words like "brown" and "cow". The intuition behind IDF is that the "specificity of a term can be quantified as an inverse function of the number of documents in which it occurs."[10] Since a corpus may contain thousands of documents, we usually compute the IDF in log-space, and smooth the quantity by adding a 1 in the denominator (to avoid divide by 0 errors).

$$\text{idf}(w) = \log(\text{number of documents} \,/\, (\text{df}(w) + 1)) \tag{21}$$

Now, with the components defined, we can define the tf-idf as:

$$\text{tf-idf}(w, d) = \text{tf}(w, d) \cdot \log(N \,/\, (\text{df}(w) + 1)) \tag{22}$$

where N is the number of documents in the corpus.

You will pre-compute the tf-idf for the dataset when you initialize the `TFIDFKernel`, prior to computing the kernel matrix. Given two documents x and x', we first obtain the term-frequencies for each document. Then, the kernel function can be computed as follows:

[9] https://en.wikipedia.org/wiki/Tf–idf

[10] Spärck Jones, K. (1972). "A Statistical Interpretation of Term Specificity and Its Application in Retrieval". Journal of Documentation. 28: 11–21.

- $k \leftarrow 0$
- for each distinct word w in x
  - freq $\leftarrow \text{tf}_{x'}(w)$ if $w \in x'$ else continue to next iteration
  - $k \leftarrow k + freq \cdot \text{tfidf}_x(w)$
- $K(x, x') = k$

where $\text{tf}_{x'}(w)$ indicates the frequency of that word in document x' and $\text{tfidf}_x(w)$ indicates the tf-idf of the word w for document x.

5.3 Computing the Kernel Matrix

You will pre-compute the kernel matrix and then index from it when you need to evaluate the kernel function. Prior to training, you will compute the kernel matrix of the train split X with itself. Prior to testing, you will compute the kernel matrix of the support vectors saved during fitting with the test split X' (see section 4.4 for more details). Keep in mind that the square part of the kernel matrix is symmetric. For the square part[11], you can just compute the upper triangular region of the matrix, and then use symmetry to copy these values into the lower triangular region.

You will fill in the details for computing the kernel matrix in the base `Kernel` class, since it will function the same regardless of which kernel function you use. Recall that with polymorphism, functions defined in the base class are accessible from child classes. So you can still call the `evaluate` method, and it will call the appropriate kernel function you define in the child classes.

You may wish to debug your kernel evaluations using a smaller number of documents. You can set the number of articles to be extracted from each class using the `--num-articles` command line argument. The default value is 500.

5.4 Track Kernel Computation Progress with `tqdm`

`tqdm`[12] has been included in the `requirements.txt` file. You can use it to add a smart progress bar to your code that informs you which iteration you're on, the elapsed time and estimated time left, and finally time required per iteration. To use it, just wrap an iterable (e.g. `range(iterations)`) in it as such `tqdm(iterable)`. An example progress bar is shown below:

```
13%|||||||                                  | 128/1000 [02:25<15:18,  1.05s/it]
```

Important: Before submitting your code to Gradescope, make sure to remove `tqdm` from your for loops. Including it will interfee with the autograder.

[11] During training the entire kernel matrix is square, this is not necessarily true during testing.
[12] Documentation: `https://github.com/tqdm/tqdm`

6 Miscellaneous

6.1 Python Libraries

We will be using Python 3.7.6. We are *not* using Python 2, and will not accept assignments written for this version. We recommend using a recent release of Python 3.7.x, but anything in this line (e.g. 3.x) should be fine.

For each assignment, we will tell you which Python libraries you may use. We will do this by providing a `requirements.txt` file. You should only use libraries available in the basic Python library or the `requirements.txt` file. In this and future assignments we will allow you to use `numpy`, `scipy`, and `tqdm`. Note, `nltk` is included in the `requirements.txt` for text processing in `data.py`. However, you may *not* use functions from this library (e.g. you may not use `nltk`'s implementation of tf-idf).

It may happen that you find yourself in need of a library not included in the `requirements.txt` for the assignment. You may request that a library be added by posting to Piazza. This may be useful when there is some helpful functionality in another library that we omitted. However, we are unlikely to include a library if it either solves a major part of the assignment, or includes functionality that isn't really necessary.

6.2 Grading Programming

The programming section of your assignment will be graded using an autograder in Gradescope. Your code will be run using the provided command line options, as well as other variations on these options (different parameters, data sets, etc.) The grader will consider the following aspects of your code.

1. **Exceptions:** Does your code run without crashing?
2. **Output:** Some assignments will ask you to write some data to the console. Make sure you follow the provided output instructions exactly.
3. **Accuracy:** If your code works correctly, then it should achieve a certain accuracy on each data set. While there are small difference that can arise, a correctly working implementation will get the right answer.
4. **Speed/Memory:** As far as grading is concerned, efficiency largely doesn't matter, except where lack of efficiency severely slows your code (so slow that we assume it is broken) or the lack of efficiency demonstrates a lack of understanding of the algorithm. For example, if your code runs in two minutes and everyone else runs in 2 seconds, you'll lose points. Alternatively, if you require 2 gigs of memory, and everyone else needs 10 MB, you'll lose points. In general, this happens not because you did not optimize your code, but when you've implemented something incorrectly.

6.3 Code Readability and Style

In general, you will not be graded for code style. However, your code should be readable, which means minimal comments and clear naming / organization. If your code works perfectly then you will get full credit. However, if it does not we will look at your code to determine how to allocate partial credit. If we cannot read your code or understand it, then it is very difficult to assign partial credit. Therefore, it is in your own interests to make sure that your code is reasonably readable and clear.

6.4 Code Structure

Your code must support the command line options and the example commands listed in the assignment. Aside from this, you are free to change the internal structure of the code, write new classes, change methods, add exception handling, etc. However, once again, do not change the names of the files or command-line arguments that have been provided. We suggest you remember the need for clarity in your code organization.

6.5 Knowing Your Code Works

How do you know your code really works? That is a very difficult problem to solve. Here are a few tips:

1. Use Piazza. While **you cannot share code**, you can share results. We encourage you to post your results on dev data for your different algorithms. A common result will quickly emerge that you can measure against.
2. Output intermediate steps. Looking at final predictions that are wrong tells you little. Instead, print output as you go and check it to make sure it looks right. This can also be helpful when sharing information on Piazza.
3. Debug. Find a Python debugger that you like and use it. This can be very helpful.

6.6 Debugging

The most common question we receive is "how do I debug my code?" The truth is that machine learning algorithms are very hard to debug because the behavior of the algorithm is unknown. In these assignments, you won't know ahead of time what accuracy is expected for your algorithm on a dataset. This is the reality of machine learning development, though in this class you have the advantage of your classmates, who may post the output of their code to the bulletin board. While debugging machine learning code is therefore harder, the same principles of debugging apply. Write tests for different parts of your code to make sure it works as expected. Test it out on the easy datasets to verify it works and, when it doesn't, debug those datasets carefully. Work out on paper the correct answer and make sure your code matches. Don't be afraid of writing your own data for specific algorithms as needed to test out different methods. This process is part of learning machine learning algorithms and a reality of developing machine learning software.

7 What to Submit

You will need to create an account on gradescope.com and signup for this class. The course is `https://www.gradescope.com/courses/153788`.

Submit your code (.py files) to gradescope.com as a zip file. **Your code must be uploaded as code.zip with your code in the root directory**. By 'in the root directory,' we mean that the zip should contain `*.py` at the root (`./*.py`) and not in any sort of substructure (for example `hw4/*.py`). One simple way to achieve this is to zip using the command line, where you include files directly (e.g., `*.py`) rather than specifying a folder (e.g., `hw4`).

8 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/kdfbbwz3hwb3an`.