**K-Nearest Neighbours for Binary Classification (50 points)**

**General instructions**

* In this task you will implement the **K-Nearest Neighbours** algorithm. We provide the bootstrap code and you are expected to complete the classes and functions.
* Do not import libraries other than those already imported in the original code.
* Please follow the type annotations. You have to make the function’s return values match the required type.
* Only modifications in files {knn.py, utils.py} in the "work" directory will be accepted and graded. All other modifications will be ignored. You can work directly on Vocareum, or download all files from "work", code in your own workspace, and then upload the changes (recommended).
* Click the Submit button when you are ready to submit your code for auto-grading. Your final grade is determined by your **last** submission.

**Background**

In this task, we will use three different functions to measure the distance of two points xx, x′∈ℝnx′∈Rn:

* Euclidean distance:

d(x,x′)=‖x−x′‖2=∑i=1n(xi−x′i)2‾‾‾‾‾‾‾‾‾‾‾‾‾⎷d(x,x′)=‖x−x′‖2=∑i=1n(xi−xi′)2

* Minkowski distance:

d(x,x′)=‖x−x′‖3=(∑i=1n∣∣xi−x′i∣∣3)1/3d(x,x′)=‖x−x′‖3=(∑i=1n|xi−xi′|3)1/3

* Cosine distance:

d(x,x′)={1,1−⟨x,x′⟩‖x‖2‖x′‖2.if ‖x‖2=0 or ‖x′‖2=0elsed(x,x′)={1,if ‖x‖2=0 or ‖x′‖2=01−⟨x,x′⟩‖x‖2‖x′‖2.else

To measure the performance of the algorithm, we will use a widely-used metric called **F1-score** (instead of the "error rate" discussed in the class). You need to self-learn the formula of F1-score from [wikipedia](https://en.wikipedia.org/wiki/F1_score" \t "_blank). Note that in this task, a "positive" sample is labeled as "1", and a "negative" one is labeled as "0".

**Part 1.1 F1 score and distance functions**

Follow the notes from Background and implement the following items in *util.py*

- f1\_score

- class Distance

- euclidean\_distance

- minkowski\_distance

- cosine\_similarity\_distance

**Part 1.2 KNN class**

Based on what we discussed in the lecture as well as the comments in the code, implement the following items in *knn.py*

- class KNN

- train

- get\_k\_neighbors

- predict

**Part 1.3 Data transformation**

We are going to add one more step (data transformation) in the data processing part and see how it works. Sometimes, normalization plays an important role to make a machine learning model work. This link might be helpful <https://en.wikipedia.org/wiki/Feature_scaling>. Here, we take two different data transformation approaches.

**Normalizing the feature vector**

This one is simple but sometimes may work well. Given a feature vector x, the normalized feature vector is given by x′=x‖x‖2x′=x‖x‖2. If a vector is an all-zero vector, we let the normalized vector to be itself.

**Min-max scaling for each feature**

The above normalization is independent of the rest of the data. On the other hand, **min-max normalization** scales each sample in a way that depends on the rest of the data. More specifically, for each feature, we normalize it linearly so that its value are between 0 and 1 across all samples, and in addition, the largest value becomes exactly 1 while the smallest becomes exactly 0. For more information and examples, read the comments in the code.

You need to implement the following items in *util.py*:

- class NormalizationScaler

- \_\_call\_\_

- class MinMaxScaler

- \_\_call\_\_

**Part 1.4 Hyperparameter tuning**

Hyperparameter tuning is an important step for building machine learning models. Here, we can treat the value of k, the distance function, and the data transformation schemes as the hyperparameters of the KNN algorithm. You need to try different combinations of these parameters and find the best model with the highest F1 score. Following the concrete comments in the code and implement the following functions in *util.py*:

- class HyperparameterTuner

- tuning\_without\_scaling

- tuning\_with\_scaling

**Part 1.5 Testing with *test.py***

There is nothing to implement in this part, but you can make use of the *test.py* file to debug your code and make sure that your implementation is correct. After you have completed all the classes and functions mentioned above, test.py file should run smoothly and show the following if your implementation is correct:

You can also uncomment Line 16 of *data.py*: np.random.shuffle(white), to shuffle the data and further test your code.

**Grading guideline (50 points)**

* F-1 score and distance functions - 15 points
* MinMaxScaler and NormalizationScaler - 10 points (5 each)
* Finding best parameters without scaling - 10 points
* Finding best parameters with scaling - 10 points
* Prediction of the best model - 5 points

**Regression (30 points)**

**General instructions**

* In this task you will implement **linear regression**. We provide the bootstrap code and you are expected to complete the functions.
* For simplicity, you do not need to append the column of 1’s to the feature matrix.
* Do not import libraries other than those already imported in the original code.
* Please follow the type annotations. You have to make the function’s return values match the required type.
* Only modifications in files {linear\_regression.py} in the "work" directory will be accepted and graded. All other modifications will be ignored. You can work directly on Vocareum, or download all files from "work", code in your own workspace, and then upload the changes (recommended).
* Click the Submit button when you are ready to submit your code for auto-grading. Your final grade is determined by your **last** submission.

**Part 1.1 Mean Square Error (4 points)**

Given a linear model parameter 𝑤w and a data set specified by 𝑋X and 𝑦y, compute the mean square error.

* TODO 1 Complete def mean\_square\_error(w, X, y) in linear\_regression.py

**Part 1.2 Linear Regression (4 points)**

Based on what we discussed in the lectures, implement linear regression with no regularization using a training data set (𝑋,𝑦)(X,y) and return the model parameter 𝑤w. You do not need to worry about non-invertible matrices for this part. You should use numpy inverse function directly (the whole implementation can in fact be as simple as one or two lines of code).

* TODO 2 Complete def linear\_regression\_noreg(X, y) in linear\_regression.py

Once you finish Part 1.1 and Part 1.2, you should be able to run linear\_regression\_test.py and test these two parts. Read the output, and check your dimension of 𝑤w (should be 12 in this case) and MSE for training, evaluation and testing datasets (should all be between 0.5~0.6).

**Part 1.3 Regularized Linear Regression (6 points)**

To prevent overfitting, we now add L2 regularization with a regularization parameter 𝜆λ.

* TODO 3 Complete def regularized\_linear\_regression(X, y, lambd) in linear\_regression.py

Once you finish this part, run linear\_regression\_test.py again. You should see a better MSE for the test data.

**Part 1.4 Tune the regularization parameter (10 points)**

Now try to tune the regularization parameter among the following 15 values: 2−14,2−13,…,2−1,20=12−14,2−13,…,2−1,20=1. More specifically, for each value, use the given training set and the regularized\_linear\_regression function you implemented in Part 1.3 to train a model, then use the given validation set and the mean\_square\_error function you implemented in Part 1.1 to evaluate the model. Finally return the best value corresponding to the model with the lowest mean square error.

* TODO 4 Complete def tune\_lambda(Xtrain, ytrain, Xval, yval) in linear\_regression.py

Once you finish this part, run linear\_regression\_test.py again. The best lambda happens to be 2−142−14 in this case.

**Part 1.5 Polynomial regression (6 points)**

In the lectures, we discussed polynomial regression for the one-dimensional case. Here, you will implement a simplified version of the polynomial regression for high-dimensional data, by only raising each feature to some power and ignoring "crossed" features. For example, if we have a two-dimensional feature (𝑥1,𝑥2)(x1,x2), then for a 2-degree polynomial regression, we will map this feature to (𝑥1,𝑥2,𝑥21,𝑥22)(x1,x2,x12,x22) (note that there is no "crossed" feature 𝑥1𝑥2x1x2).

To reuse previous code for linear regression, you task is simply to take a dataset 𝑋X and an integer 𝑝p, and return the augmented data set [𝑋,𝑋2,…,𝑋𝑝][X,X2,…,Xp] where 𝑋𝑖Xi stands for element-wise power.

* TODO 5 Complete def mapping\_data(X, power) in linear\_regression.py

Once you finish this part, run linear\_regression\_test.py again. You should see that the training MSE is getting smaller as we use a larger degree, but at the same time the testing MSE is increasing due to overfitting.

**Neural Nets (100 points)**

**General instructions**

* In this task you will implement **neural nets**. We provide the bootstrap code and you are expected to complete the functions.
* Do not import libraries other than those already imported in the original code.
* Please follow the type annotations. You have to make the function’s return values match the required type.
* Only modifications in files {neural\_networks.py} in the "work" directory will be accepted and graded. All other modifications will be ignored. You can work directly on Vocareum, or download all files from "work", code in your own workspace, and then upload the changes (recommended).
* Click the Submit button when you are ready to submit your code for auto-grading (**read Q7 carefully before doing so**). Your final grade is determined by your **last** submission.

**Neural Networks**

In this task, you are asked to implement neural networks. You will use this neural network to classify MNIST database of handwritten digits (0-9). The architecture of the neural network you will implement is based on the multi-layer perceptron (MLP, just another term for fully connected feedforward networks we discussed in the lecture), which is shown in Figure 1. It is designed for a K-class classification problem. Let (x∈ℝD,y∈{1,2,⋯,K})(x∈RD,y∈{1,2,⋯,K}) be a labeled instance, such an MLP performs the following computations:

**input features**:**linear**(1):**tanh**:**relu**:**linear**(2):**softmax**:**predicted label**:x∈ℝDu=W(1)x+b(1),W(1)∈ℝM×D and b(1)∈ℝMh=21+e−2u−1h=max{0,u}=⎡⎣⎢⎢⎢max{0,u1}⋮max{0,uM}⎤⎦⎥⎥⎥a=W(2)h+b(2),W(2)∈ℝK×M and b(2)∈ℝKz=⎡⎣⎢⎢⎢⎢⎢⎢⎢ea1∑keak⋮eaK∑keak⎤⎦⎥⎥⎥⎥⎥⎥⎥ŷ =argmaxkzk.input features:x∈RDlinear(1):u=W(1)x+b(1),W(1)∈RM×D and b(1)∈RMtanh:h=21+e−2u−1relu:h=max{0,u}=[max{0,u1}⋮max{0,uM}]linear(2):a=W(2)h+b(2),W(2)∈RK×M and b(2)∈RKsoftmax:z=[ea1∑keak⋮eaK∑keak]predicted label:y^=argmaxkzk.

**Q1. Linear Layer**

First, you need to implement the linear layer of an MLP by implementing 3 python functions in class linear\_layer. This layer has two parameters WW and bb.

* In the function def \_\_init\_\_(self, input\_D, output\_D), you need to randomly initialize the entries of WW and bb with mean 0 and standard deviation 0.1 using np.random.normal. You also need to initialize the gradients to zeroes in the same function.
* In def forward(self, X), implement the forward pass of this layer. Note that the input XX contains several examples, each of which needs to be passed through this layer. Try to use matrix operation instead of for loops to speed up the code.
* In def backward(self, X, grad), implement the backward pass of this layer. Here, grad is the gradient with respect to the output of this layer, and you need to find and store the gradients with respect to WW and bb, and also find and return the gradients with respect to the input XX of this layer.

**Q2. Activation function - RELU**

Next, you need to implement the RELU activation by implementing 2 python functions in class relu. There are no parameters to be learned in this module.

* In def forward(self, X), implement the forward pass of RELU activation.
* In def backward(self, X, grad), implement the backward pass of RELU activation. Here, grad is the gradient with respect to the output of this layer, and you need to return the gradient with respect to the input X.

**Q3. Activation function - tanh**

Next, you need to implement another activation function tanh, by implementing 2 python functions in class tanh. There are no parameters to be learned in this module.

* In def forward(self, X), implement the forward pass of tanh activation.
* In def backward(self, X, grad), implement the backward pass of tanh activation. Here, grad is the gradient with respect to the output of this layer, and you need to return the gradient with respect to the input X. See comments in the code for the formula of the derivative of tanh.

**Q4. Dropout**

Dropout is one effective way to prevent overfitting. You will implement one python function in class dropout to implement dropout.

* In def forward(self, X, is\_train), we have implemented the forward pass of dropout when is\_train is true (since we do not perform dropout in testing). It executes the following operation:

forward pass:s=dropout.forward(q∈ℝJ)=11−r×⎡⎣⎢⎢⎢**1**[p1>=r]×q1⋮**1**[pJ>=r]×qJ⎤⎦⎥⎥⎥,where pj is generated randomly from [0,1),∀j∈{1,⋯,J},and r∈[0,1) is a pre-defined scalar named dropout rate which is given to you.forward pass:s=dropout.forward(q∈RJ)=11−r×[1[p1>=r]×q1⋮1[pJ>=r]×qJ],where pj is generated randomly from [0,1),∀j∈{1,⋯,J},and r∈[0,1) is a pre-defined scalar named dropout rate which is given to you.

You only need to read and understand the code here (i.e, nothing for you to implement).

* In def backward(self, X, grad), implement the backward pass of dropout, which performs the following operation:

backward pass:∂l∂q=dropout.backward(q,∂l∂s)=11−r×⎡⎣⎢⎢⎢⎢⎢⎢⎢**1**[p1>=r]×∂l∂s1⋮**1**[pJ>=r]×∂l∂sJ⎤⎦⎥⎥⎥⎥⎥⎥⎥.backward pass:∂l∂q=dropout.backward(q,∂l∂s)=11−r×[1[p1>=r]×∂l∂s1⋮1[pJ>=r]×∂l∂sJ].

Note that p1,…,pJp1,…,pJ should not be sampled randomly again but take the same values as in the forward pass.

**Q5. Mini-batch Stochastic Gradient Descent**

Next, implement a mini-batch version of stochastic gradient descent with momentum to learn the parameters of the neural net. Recall that for a general optimization problem with a parameter ww, SGD with momentum iteratively computes the following

υt=αυt−1−ηgtwt=wt−1+υtυt=αυt−1−ηgtwt=wt−1+υt

where αα is the momentum parameter, ηη is the step size, and gtgt is the stochastic gradient (at wt−1wt−1). You need to complete def miniBatchStochasticGradientDescent(model, momentum, \_alpha, \_learning\_rate), where we update the parameters of each layer. Note that this function is executed after the backward pass of each layer has been called and thus the gradients have been stored properly. In fact, for each parameter, we have included the code to find its gradient, and you only need to do the following:

* if α≤0α≤0, implement one step of gradient descent without momentum, with the given step size and gradient;
* if α>0α>0, implement one step of gradient descent with momentum, with the given step size, momentum parameter, and gradient.

**Q6. Connecting the dots**

We connect all the parts in the function main(main\_params). Read the code carefully and see how different parts are connected together. The only thing you need to implement here is to call the backward pass of each layer in the correct order and with the correct inputs.

**Q7. Testing your code**

After completing all the above, you should be able to run neural\_networks.py, which tests your code on a subset of the MNIST dataset for 10 epochs. If your implementation is correct, you should be seeing something similar to the below (only showing the first 4 epochs; it is okay if the numbers are slightly different). The training accuracy and validation accuracy are also stored in a file called MLP\_lr0.01\_m0.0\_d0.5\_arelu.json.

**IMPORTANT:** before submitting this task for grading, you have to run runme.py. This runs your code on the same MNIST dataset but with 4 different sets of hyperparameters, and produces four json files:

* MLP\_lr0.01\_m0.0\_d0.0\_arelu.json
* MLP\_lr0.01\_m0.0\_d0.0\_atanh.json
* MLP\_lr0.01\_m0.9\_d0.25\_atanh.json
* MLP\_lr0.01\_m0.9\_d0.5\_arelu.json

Then you have to **upload** these four json files to the **work** directory (same as your neural\_networks.py file). If you run runme.py directly on the Vocareum's console, then these files are in the **work** directory already. Once you confirm that you have these files in addition to your modified neural\_networks.py in this directory, you can click submit for auto-grading.

**Grading guideline**

* Linear layer: 15 points
* Activation function - relu: 15 points
* Activation function - tanh: 15 points
* Dropout:15 points
* Four test cases: 40 points (10 each)

**Linear Classifiers (60 points)**

**General instructions**

* In this task you will implement **linear classifiers** by minimizing perceptron loss or logistic loss. We provide the bootstrap code and you are expected to complete the functions.
* Do not import libraries other than those already imported in the original code.
* Please follow the type annotations. You have to make the function’s return values match the required type.
* Only modifications in files {bm\_classify.py} in the "work" directory will be accepted and graded. All other modifications will be ignored. You can work directly on Vocareum, or download all files from "work", code in your own workspace, and then upload the changes (recommended).
* Click the Submit button when you are ready to submit your code for auto-grading. Your final grade is determined by your **last** submission.

**Part 1. Binary Classification - Perceptron vs. Logistic**

In this part you are given a training set for a binary classification problem, and your task is to train a linear classifier. To test if you truly understand the algorithms discussed in the class, we intensionally introduce some extra complexity: first, the labels are either 0 or 1, instead of -1 or +1; second, the features do not contain the constant value 1, and you do need to explicitly learn a bias term (in other words, you are trying to learn a model 𝑤⊤𝑥+𝑏w⊤x+b, instead of just 𝑤⊤𝑥w⊤x). These extra complications are often met in practice, and you need to think carefully before applying the formulas from the lectures.

* TODO 1 In function binary\_train, implement Gradient Descent to minimize the \*average\* perceptron loss defined in terms of the given training set. You are given the initial parameters, the step size, and the number of iterations needed.
* TODO 2 In function binary\_train, implement Gradient Descent to minimize the \*average\* logistic loss defined in terms of the given training set. Again, you are given the initial parameters, the step size, and the number of iterations needed. It will be convenient to implement the sigmoid function sigmoid for this part, so please do so in TODO 3.
* TODO 4 In function binary\_predict, implement the prediction of a given linear model on a given test set. Note that your prediction should be deterministic (that is, do not randomize).

After you finish this part, you can run binary.sh to test your code, which produces a file binary.out. If your code is programmed correctly, you should see something similar to below (it is okay if the numbers are not exactly the same but close nough).

Below is the visualization of the first two datasets:

* Synthetic data:
* Two Moon data:

**Part 2. Multiclass classification - SGD vs. GD**

Next, you need to implement multinomial logistic regression with either Gradient Descent or Stochastic Gradient Descent for a multiclass classification problem. Again, please note the following: first, the labels are indexed by 0,1,…,𝐶−10,1,…,C−1; second, you need to explicitly learn the bias vector.

In addition, please follow the following tip when implementing the softmax function (an essential part of the algorithm), to avoid numerical issues such as overflow and underflow caused by np.exp. Suppose the original input vector to the softmax function is 𝑧z. In the actual implementation, use vector 𝑧̃ =𝑧−max(𝑧)z~=z−max(z) as the input of the softmax function instead. That is, if you want to compute exp(𝑧𝑖)∑𝐷𝑗=1exp(𝑧𝑗)exp⁡(zi)∑j=1Dexp⁡(zj), compute exp(𝑧̃ 𝑖)∑𝐷𝑗=1exp(𝑧̃ 𝑗)exp⁡(z~i)∑j=1Dexp⁡(z~j) instead, which is clearly mathematically equivalent but numerically more stable.

* TODO 5 In function multiclass\_train, implement Stochastic Gradient Descent to minimize the \*average\* multiclass logistic loss defined in terms of the given training set. You are given the initial parameters, the step size, and the number of iterations needed. Moreover, in each iteration of SGD, we have already randomly selected a sample for you (indexed by 𝑛n). Please do not modify this part or the random seed in your final submission (you can of course change it when debugging your code if needed).
* TODO 6 In function multiclass\_train, implement Gradient Descent to minimize the \*average\* multiclass logistic loss defined in terms of the given training set. Again, you are given the initial parameters, the step size, and the number of iterations needed. Try to use matrix calculation provided in the numpy package instead of nested for loops in your implementation, which great reduces the running time.
* TODO 7 In function multiclass\_predict, implement the prediction of a given linear model on a given test set. Again, your prediction should be deterministic (that is, do not randomize).

After you complete this part, run multiclass.sh to test your code, which produces a file multiclass.out. If your code is programmed correctly, you should see something similar to the below (it is okay if the numbers are not exactly the same but close enough).

Below is the visualization of the first toy dataset:

**Grading guideline**

* Sigmoid function - 4 points
* Binary classification - 24 points (4 test cases)
* Multiclass classification - 32 points (4 test cases)

Note that the first test case for binary classification is the Two Moon Dataset you can play with when running binary.sh. To help you debug your code, in the grading report we have included the expected outputs for this test case.

Another important note: if your code takes too long to run, Vocareum will kill the process automatically. You are responsible to make sure that your code can be graded successfully on Vocareum. As a reference, it takes less than one minute for our solution code to be graded.

**K-means Clustering (50 points)**

**General instructions**

* In this task you will implement the **K-means** algorithm and its applications. We provide the bootstrap code and you are expected to complete the functions.
* Do not import libraries other than those already imported in the original code.
* Please follow the type annotations. You have to make the function’s return values match the required type.
* Only modifications in files {kmeans.py} in the "work" directory will be accepted and graded. All other modifications will be ignored. You can work directly on Vocareum, or download all files from "work", code in your own workspace, and then upload the changes (recommended).
* Click the Submit button when you are ready to submit your code for auto-grading. Your final grade is determined by your **last** submission.

**High Level Description**

In this assignment you are asked to implement:

* the K-means++ initialization,
* the standard K-means algorithm given the initialization,
* the nearest neighbor algorithm using the K-means centroids,
* image compression via the K-means algorithm.

NOTE: Depending on your environment you may need to install the python library named "pillow", which is used by matplotlib to process some of the images needed for this assignment. You can install it by running "pip3 install pillow" in your command line.

**Q1. K-means++ initialization**

K-means++ generally performs much better than the vanilla K-means algorithm. The only difference is in the initialization of the centroids. According to the discussions in the lecture, implement this initialization in function get\_k\_means\_plus\_plus\_center\_indices. (Note that we also provide the vanilla initialization method in get\_lloyd\_k\_means.)

**Q2. K-means algorithm**

Recall that for a dataset x\_1, . . . , x\_N ∈ R^Dx1​,...,xN​∈RD, the K-means distortion objective is:  
F(\{\mu\_k\}, \{r\_{nk}\}) = \sum\_{i=1}^N \sum\_{k=1}^K r\_{nk} \|\mu\_k- x\_n\|\_2^2 \qquad (1)J({μk​},{rnk​})=n=1∑N​k=1∑K​rnk​∥μk​−xn​∥22​(1)

where µ\_1, . . . , µ\_Kµ1​,...,µK​ are centroids of the K clusters and r\_{nk} ∈ {0, 1}rnk​∈{0,1} represents whether example n belongs to cluster k.  
  
In this part, you need to implement the K-means procedure that iteratively computes the new cluster centroids and assigns data points to the new clusters. The procedure stops whenever 1) the number of updates has reached the given maximum number, or 2) when the \*average\* K-means distortion objective J changes less than a given threshold between two iterations.  
  
Implement this part in the fitfunction of the class KMeans. While assigning a sample to a cluster, if there is a tie (i.e. the sample is equidistant from two or more centroids), you should choose the one with the smaller index (which is what numpy.argmin does already).

After you complete the implementation, run KmeansTest.py to see the results of this on a toy dataset. You should be able to see three images generated in a folder called plots. In particular, you can see toy\_dataset\_predicted\_labels.png and toy\_dataset\_real\_labels.png, and compare the clusters identified by the algorithm against the real clusters. Your implementation should be able to recover the correct clusters sufficiently well. Representative images are shown below. Red dots are cluster centroids. Note that color coding of recovered clusters may not match that of correct clusters. This is due to mis-match in ordering the retrieved clusters and the correct clusters (which is fine).

**Q3 Classification with K-means**

Another application of clustering is to obtain a faster version of the nearest neighbor algorithm. Recall that nearest neighbor evaluates the distance of a test sample from every training point to predict its label, which can be very slow. Instead, we can compress the entire training dataset to just K centroids, where each centroid is now labeled as the majority class of the corresponding cluster. After this compression the prediction time of nearest neighbor is reduced from O(N) to just O(K) (see below for the pseudocode).  
  
  
  
You need to complete the fit and predict function in KMeansClassifier following the comments in the code. Again, whenever you need to break a tie, pick the one with the smallest index.  
  
Once completed, run KmeansTest.py again to evaluate the classifier on a test set (digits). For comparison, the script will also print accuracy of a logistic classifier and a vanilla nearest neighbor classifier. An example is shown below.

**Q4 Image compression with K-means**

In this part, we will take lossy image compression as another application of clustering. The idea is simply to treat each pixel of an image as a point, then perform K-means algorithm to cluster these points, and finally replace each pixel with its closest centroid.

What you need to implement is to compress an image with K centroids given (called code\_vectors). Specifically, complete the function transform\_image following the comments in the code.  
  
After your implementation, run KmeansTest.py again. You should be able to see an image compressed\_baboon.png in the plots folder. You can see that this image is slightly distorted as compared to the original baboon.tiff. The ideal result should take about 35-40 iterations and the Mean Square Error (between the two images) should be less than 0.0098. It takes about 1-2 minutes to complete normally.

**Grading Guidelines:**

* get\_k\_means\_plus\_plus\_center\_indices - 5 points (5 test cases)
* Kmeans class - 15 points (5 test cases)
* KmeansClassifier class - 20 points (4 test cases)
* transform\_image - 10 points (2 test cases)

Note: it takes a few minutes for the grading to finish.

**Hidden Markov Models (50 points)**

**General instructions**

* In this task you will implement various inference algorithms for **HMM** and also apply them to sentence tagging. We provide the bootstrap code and you are expected to complete the functions.
* Do not import libraries other than those already imported in the original code.
* Please follow the type annotations. You have to make the function’s return values match the required type.
* Only modifications in files {hmm.py, tagger.py} in the "work" directory will be accepted and graded. All other modifications will be ignored. You can work directly on Vocareum, or download all files from "work", code in your own workspace, and then upload the changes (recommended).
* Click the Submit button when you are ready to submit your code for auto-grading. Your final grade is determined by your **last** submission.

**Q1 Implement the inference algorithms**

In hmm.py, you will find a class called HMM whose attributes specify the model parameters of a Hidden Markov Model (including its initial state probability, transition probability, and emission probability). You need to implement the following six functions

* forward: compute the forward messages
* backward: compute the backward messages
* sequence\_prob: compute the probability of observing a particular sequence
* posterior\_prob: compute the probability of the state at a particular time step given the observation sequence
* likelihood\_prob: compute the probability of state transition at a particular time step given the observation sequence
* viterbi: compute the most likely hidden state path using the Viterbi algorithm.

We have discussed how to compute all these via dynamic programming in the lecture. Here, the only thing you need to pay extra attention to is that the indexing system is slightly different between the python code and the formulas we discussed (the former starts from 0 and the latter starts from 1). Read the comments in the code carefully to get a better sense of this discrepancy.

**Q2 Application to speech tagging**

Part-of-Speech (POS) is a category of words (or, more generally, of lexical items) which have similar grammatical properties. (Example: noun, verb, adjective, adverb, pronoun, preposition, conjunction, interjection, and sometimes numeral, article, or determiner.) Part-of-Speech Tagging (POST) is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context.

Here you will use HMM to perform POST, where the tags are states and the words are observations. We collect our dataset and tags with the Dataset class. Dataset class includes tags, train\_data and test\_data. Both datasets include a list of sentences, and each sentence is an object of the Line class. You only need to implement the model\_training function and the speech\_tagging function in tagger.py.

* model\_training: in this function, you need to build an instance of the HMM class by setting its five parameters (pi, A, B, obs\_dict, state\_dict). The way you estimate the parameter pi, A, B is simply by counting the corresponding frequency from the given training set, as we discussed in the class. Read the comments in the code for more instructions.
* speech\_tagging: given the HMM built from model\_training, now your task is to run the Viterbi algorithm to find the most likely tagging of a given sentence. One particular detail you need to take care of is when you meet a new word which was unseen in the training dataset. In this case, you need to update the dictionary obs\_dict accordingly, and also expand the emission matrix by assuming that the probability of seeing this new word under any state is 1e-6. Again, read the comments in the code for more instructions.

**Q3 Testing**

Once you finish these two parts, run hmm\_test\_script.py. We will first run all your inference algorithms on a toy HMM model specified in hmm\_model.json, and then also your tagging code on the dataset stored in pos\_sentences.txt and pos\_tags.txt. In both cases, the script tells you what your outputs vs. the correct outputs are.

**Grading guideline**

1 Inference algorithms (30 points)

1. forward function - 5 = 5x1 points
2. backward function - 5 = 5x1 points
3. sequence\_prob function - 2.5 = 5x0.5 points
4. posterior\_prob function - 5 = 5x1 points
5. likelihood\_prob function - 5 = 5x1 points
6. viterbi function - 7.5 = 5\*1.5 points

There are 5 sets of grading data used to initialize the HMM class and test your functions. To receive full credits, your output of functions 1-5 should be within an error of 1e-6, and your output of the viterbi function should be identical with ours.

2 Application to Part-of-Speech Tagging (20 points)

1. model\_training - 10 = 10x(your\_correct\_pred\_cnt/our\_correct\_pred\_cnt)
2. speech\_tagging - 10 = 10x(your\_correct\_pred\_cnt/our\_correct\_pred\_cnt)

We will use the dataset given to you for grading this part (with a different random seed). We will train your model and our model on same train\_data. model\_training function and speech\_tagging function will be tested separately.

In order to check your model\_training function, we will use 50 sentences from train\_data to do Part-of-Speech Tagging (your model + our tagging function vs. our model + our tagging function). To receive full credits, your prediction accuracy should be identical or better than ours.

In order to check your speech\_tagging function, we will use 50 sentences from test\_data to do Part-of-Speech Tagging (your model + your tagging function vs. our model + our tagging function). Again, to receive full credits, your prediction accuracy should be identical or better than ours.