Machine Learning for Molecular Engineering
3/7/10/20.C01 (U) 3/7/10/20.C51 (G)
Spring 2024

Problem Set #4 - SMILES

Date: March 21, 2024
Due: April 11, 2024 @ 11:59 pm

Instructions

- **Important:** This problem set requires a GPU. Before you start in Google Colab, find `Notebook Settings` under the `Edit` menu. In the `Hardware accelerator` drop-down, select a GPU as your hardware accelerator.
- This problem set contains two questions, each containing sub-questions. Some questions have sub-questions marked as grad version. These questions are mandatory for those who have registered for the graduate version of this class (X.C51). They are optional for others. We've listed the points associated with each question in blue for the undergraduate and graduate version and orange for the graduate-only version. The undergraduate version is out of 75 and the graduate version is out of 100 points.
- Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `*.ipynb`. Your submission should contain *all* of the code that is needed to fully answer the questions in this problem set when executed from top to bottom and should run without errors (unless intentional). Please ensure that your plots are visible in the output file and that you are printing any additional comments or variables as needed. Commenting your code in-line and adding any additional comments in markdown (as "Text cells") will help the grader understand your approach and award partial credit.
- Note that collaboration between students is encouraged, however every student must be responsible for all the work in their individual submission. I.e., discussing solutions and debugging together should be considered fruitful teamwork, however physically taking the keyboard of another student and directly entering code for their submission would be intellectually dishonest and as such is prohibited. You should first read through and attempt every problem before discussing with others. Please list the names of all collaborators at the bottom of your .ipynb submission file.
- To get started, open your Google Colab or Jupyter notebook starting from the problem set template file pset4_bio_template.ipynb (direct Colab link). You will need to use the data files here.

Part 1: Dimensionality Reduction for Molecular Representations

In this part, you will be performing dimensionality reduction analysis on a set of candidate molecules tested against the Mitogen-Activated Protein Kinase, also called MAPK1 or ERK2. The data is from the DUD-E database.

Converting the complex structure and topology of a molecule into a vector representation is not straightforward. The cheminformatics community has developed a variety of algorithms to numerically encode the structural information of molecules. Morgan fingerprints, also known as circular fingerprints, represent the molecular graph as a unique bit string that encodes the molecular topology. You can understand it as a bit-vector categorical encoding with 1 for substructures present in the molecule and 0 for those not present. Some implementations of fingerprints use counts of how many times a substructure is present, instead of just 0s and 1s for absence/presence. Morgan fingerprints are typically high dimensional to maximize their descriptive power and sizes of 512, 1024 or 2048 are common (you can choose the number of bits when generating these fingerprints with the Morgan Algorithm). Morgan fingerprints provide a vector representation of molecules that is nearly unique or each molecule (although fingerprint collisions for non-identical molecules are possible), on which we can apply machine learning algorithms.

In this question, you will apply dimensionality reduction algorithms on a high-dimensional molecular fingerprint data we provided.

Part 1.1 (5 points, Grad only) Choosing radius and number of bits for Morgan fingerprints

You will notice that we set the radius and number of bits to 3 and 512, respectively, in the code we provided to convert SMILES to Morgan fingerprints. Do some quick researching on the radius and number of bits parameters in in RDKit's Morgan fingerprinting. Provide a one-sentence description of what the radius represents and another of what the number of bits represents. How does adjusting the radius parameter affect the granularity of the motifs captured by the fingerprints, and how does this relate to the choice of the number of bits? (No code, just your thoughts for this.)

Part 1.2 (10 points) Principal Component Analysis on Molecular Fingerprints

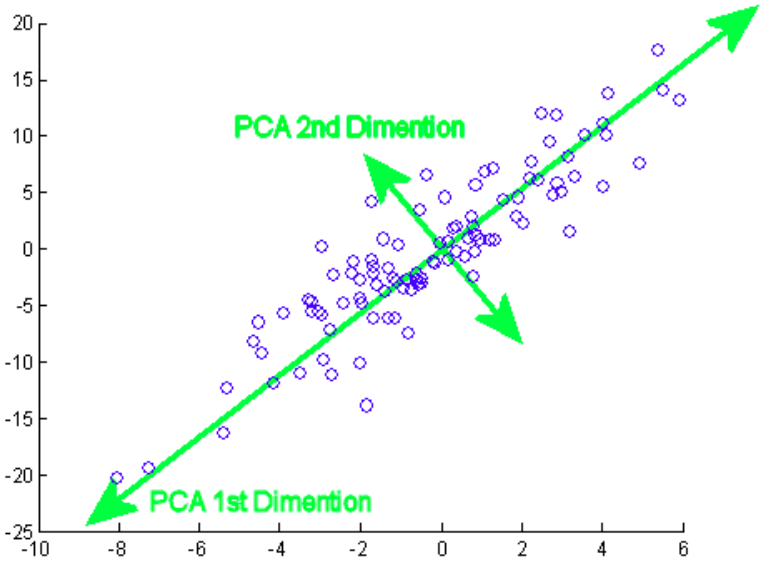

Figure 1: An example PCA analysis on a 2D dataset. The principal components are marked in green arrows. (source)

First load the fingerprint data arrays of 512 bit size and the molecule dataframe with the code we

provided to you (The row in the fingerprint array is mapped one-to-one to the row of the molecule dataframe). The fingerprint is computed with the Morgan algorithm implemented in the RDkit package. In the dataframe, we also provided a binary label to indicate if the molecule tested active against the target protein.

Each molecular fingerprint spans a 512-dimensional space, so it is hard to make sense of it. We encourage you to think about the size of the space that can be encoded in such representation.

Principal Component Analysis (PCA) is a dimensionality-reduction method that is often used to tackle high-dimensional, large data sets. In PCA, a large set of variables is transformed into a smaller one that contains as much of the information in the larger set as possible. PCA performs a linear transformation on your data by re-projecting in onto a new orthogonal basis. These new basis is chosen such that the fewest basis vectors best explain the variance of your data. Because of ease of visualization, it is common to just interrogate the first two dimensions (also called eigenvectors or components) of the PCA, but its possible to select additional dimensions.

You should perform PCA on the fingerprint data with `sklearn.decomposition.PCA()`. Use 100 components by setting `n_components=100`. This will reduce your original 512 dimensional space onto just a space of just 100 dimensions. Visualize the first two components of your data in a 2D scatter plot and color the active and inactive molecules differently. What is the total explained variance ratio of the first 100 principal components? (This is an estimate of how much information is preserved after this dimensionality reduction procedure) You can retrieve the ratios of explained variances by checking the `.explained_variance_ratio_` attribute of a `PCA()` object.

Inspect the scatter plot. Do you observe any patterns in the distribution of active drugs in this 2D plot?

Part 1.3 (10 points) t-SNE analysis on Molecular Fingerprints

PCA applies linear transformations on your data, but may struggle to tease out subtle underlying patterns in your data. Non-linear transformations can distort or compress the dimensions much more effectively and can uncover much more complex structure in the dataset. There are two popular choices for non-linear dimension reduction: UMAP and t-SNE. t-Distributed Stochastic Neighbor Embedding (t-SNE) performs a stochastic non-linear transformation on your data and it is implemented in sklearn ((`sklearn.manifold.TSNE()`)). In general, t-SNE is a random algorithm so you will have different looking embeddings every time; it is also sensitive to *perplexities* which balance attention between local and global aspects of your data during the optimization. Read the documentation for `TSNE()` to understand these options.

Apply t-SNE on your the 100-dimensional embedding you obtained from previous part with `n_components=2` with a perplexity value of 2, 30 and 500 separately and produce three labeled t-SNE plots with active and inactive drugs in different colors. The t-SNE computation might take a few minutes. Do you notice any differences in the clustering? Focus on the plots obtained with `perplexity=30`. In your 2D scatter plot, do you observe any patterns for the active drugs?

Part 1.4 Graduate (20 points) Are the low dimensional embeddings meaningful?

In this question, you will be asked to make sense of the extracted low dimensional embeddings and how the may relate to prediction performance in a supervised learning task. You will need to train a Random Forest classifier of active vs. inactive compounds and then relate the performance of this supervised to the embeddings you learned in the previous part.

Split your data into 10 folds using `sklearn.model_selection.KFold()` and train a random forest classifier with 100 trees (`n_estimators=100`). Use the full fingerprint (not the low-dimensional embedding) as your featurization for the random forest. Just as you did in PSet 1, use `sklearn.ensemble.RandomFo` Test each fold with models trained on the other 9 folds, and record your prediction in a dataframe or an array. Classify your predictions in 4 categories: True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). Make a scatter plot with the t-SNE embeddings, and color each of the four categories above with different colors. Inspect your plot; do you observe any patterns in the prediction categories (TP, TN, FP, FN) on your t-SNE plots?

Part 2: Variational auto-encoders (VAEs) for SMILES strings

This part is completely independent of Part 1 and can be done in a separate Colab session.

Variational Auto-Encoders (VAEs) are a class of generative models. The encoder consists of Gated Recurrent Units to encode a SMILES string into a latent representation. The decoder is also a stacked GRU that takes the latent representation to reconstruct the input SMILES. The autoencoder is trained to use the encoder and decoder to reconstructs your input as closely as possible. In this exercise, you will be asked to implement the sampling and loss function for a SMILES VAE. Because training a VAE on a large dataset can be computationally demanding, we have provided a SMILES-VAE model that is pre-trained on 1 million molecules. You can load the model with the code we provide. You will train on a smaller dataset of of 50,000 molecules to fine-tune the model. It is still a lot of data, so please train your model on a GPU.

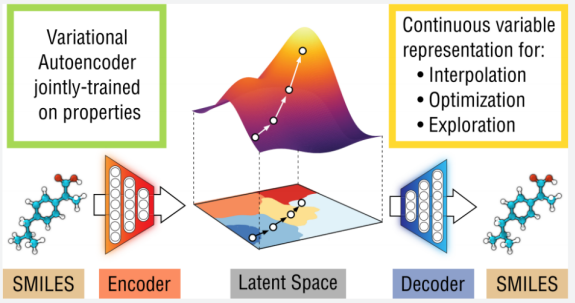

Figure 2: Applying a VAE on SMILES strings for molecular design [1]

Part 2.1 (5 points) One-hot encode SMILES strings into padded numerical vectors

As introduced in the background section, molecules can be represented as 1D strings with SMILES representation. Encode each character in a SMILES string into categorical numbers (note that this is *not* the same as one-hot encoding) using `LabelEncoder()` from `sklearn`. We have provided a list of string characters in `moses_charset` which you can use as a dictionary to write a SMILES. This dataset has SMILES strings with different character length, this requires an additional preprocessing procedures called padding, *i.e.*, adding empty characters (`' '`) to the end of all strings to make sure all the strings has the same length. The length of strings is determined by the longest SMILES string which you need to find out.

Next, transform your encoded SMILES data into a `torch.LongTensor`. This is an unsupervised learning task, so there is no `y`. Split your data into a 60:20:20 train:validation:test split as before,

using `torch.utils.data.TensorDataset` as your `Dataset` object. Like what you did in Problem Set 3, use a `DataLoader` with `batch=512` and `shuffle=True`.

Part 2.2 (15 points) Implement the reparametrization trick for VAE

We have provided the implementation for a SMILES-VAE in the `MoleVAE()` class. In VAEs, the encoding of an input x into an embedding z is not deterministic. The model is trying to compress the data into a latent distribution via a conditional distribution $Q_\phi(z|x) = \mathcal{N}(\mu_\phi, \sigma_\phi^2)$ which is parametrized by an encoder function, ϕ. The model needs to sample from the distribution: $z \sim Q_\phi(z|x)$. However, this sampling process requires some extra attention because we want to ensure the sampling procedure is differentiable for gradient optimization. Generally, we cannot do this for a sampling process:

$$z \sim \mathcal{N}(\mu_\phi, \sigma_\phi^2) \tag{1}$$

since the derivatives $\frac{dz}{d\mu_\phi}$ and $\frac{dz}{d\sigma_\phi}$ are not clearly defined for this sampling procedure. However, we can use the reparameterization trick which suggests that we randomly sample ϵ from a unit multivariate Gaussian, and then shift the randomly sampled ϵ by the latent distribution's mean μ and scale it by σ.

$$\begin{aligned} \epsilon &\sim \mathcal{N}(0, \mathbf{1}) \\ z &= \mu_\phi + \epsilon \cdot \sigma_\phi \end{aligned} \tag{2}$$

The reparametrization trick makes the sampling process from a multivariate Gaussian *differentiable*! The reason why it works is because the randomness in the sampling is 'reparameterized' into a leaf node which does not require gradient calculation in the backward computation.[1] To put it more concretely, during the backpropagation process when the gradient $\frac{dL}{dz}$ is computed (L is the scalar loss function), the gradients on μ_ϕ and σ_ϕ can be further computed with:

$$\begin{aligned} \frac{dL}{d\mu_\phi} &= \frac{dL}{dz}\frac{dz}{d\mu_\phi} = \frac{dL}{dz} \\ \frac{dL}{d\sigma_\phi} &= \frac{dL}{dz}\frac{dz}{d\sigma_\phi} = \frac{dL}{dz}\epsilon \end{aligned} \tag{3}$$

The distributional output of the encoder input requires additional attention. Note that the encoder module parametrized by an MLP might output a negative σ_ϕ^2 which might annoy a statistician. The numerical trick to ensure that σ_ϕ has only positive values is to output the log of σ_ϕ^2 instead and then exponentiate:

$$\begin{aligned} \mu_\phi, \log\sigma_\phi^2 &= \texttt{encoder(SMILES)} \\ \sigma_\phi &= \exp\left(\frac{1}{2}\log\sigma_\phi^2\right) \end{aligned} \tag{4}$$

First implement the function to transform $\log\sigma_\phi^2$ to σ in `MolVAE().get_std()` (as in equation 4)). Then, implement the reparametrization trick in the `MolVAE().reparametrize()` function which takes two inputs: μ_ϕ and σ_ϕ and outputs a latent vector z as in equation 2 (you will need the `torch.randn` method to sample ε). Test your program `reparametrize()` by using our provided code to sample 1000 samples from a 1D distribution with $\mu = 0$ and $\sigma^2 = \mathbf{1}$ and compare the sampled distribution with $\mathcal{N}(0, \mathbf{1})$.

[1] The reparameterization trick is a useful technique for variational inference trained with a gradient-based method. Similar reparameterization tricks can be derived for other types of distributions like the Beta, Dirichlet and Von-Mises distributions [2].

Part 2.3 (10 points) Implement the SMILES VAE loss function

The decoder model $P_\theta(x|z)$, parameterized by a set of parameters θ, takes the sampled latent code $z \sim Q_\phi(z|x)$ to reconstruct x which is your input SMILES string. The training objective of a VAE is to minimize the negative evidence lower bound, or ELBO, which can be understood as optimizing a reconstruction loss and regularization term. The regularization term is the KL divergence between the parameterized distribution and the prior distribution.

$$\begin{aligned} L &= L_{recon} + \beta L_{regularization} \\ &= \int dz\, Q_\phi(z|x) \log P_\theta(x|z) + \beta \int dz\, p(z) \log \frac{p(z)}{Q_\phi(z|x)} \end{aligned} \tag{5}$$

β is a hyperparameter that balances the two loss terms (see Ref.[3] for more information about the effect of β). The reconstruction term compares the original input and the decoded inputs. The input data here is a sequence of vectors with encoded categorical values and the output is a sequence with probabilities for each character categories, so we can use `nn.Functional.cross_entropy` as the training objective to minimize.

$$L_{recon} = -\frac{1}{N_{seq} N_{char}} \sum_i^{N_{seq}} \sum_k^{N_{char}} p_{data}(\hat{x}_{i,k}) \log(p(x_{i,k})) \tag{6}$$

$\log(p(x_{i,k}))$ is the logit for each possible character category reconstructed by the decoder, $\hat{x}_{i,k}$ is the original molecular SMILES; N_{seq} is the length of the sequence and N_{char} is the total number of possible characters in the sequence. `nn.Functional.cross_entropy` takes two inputs, the predicted logits for each character at each position in the SMILES string (dimension = $N_{batch} \times N_{seq} \times N_{char}$), and the original data as character category represented by integers at each position in the padded SMILES string (dimension = $N_{batch} \times N_{seq}$)). Please check the documentation for `nn.Functional.cross_entropy` for example usage.

Warning: Check your dimensions carefully in this step. In particular, the decoder output will need the batch size to be dimension 0, the number of characters to be dimension 1, and the sequence length to be dimension 2, and the original SMILES input will need the batch size to be dimension 0 and the sequence length to be dimension 1 for `nn.Functional.cross_entropy` to work as intended. Transpose your tensor with `transpose` if necessary to get this to line up.

A simple prior distribution one can choose is a multivariate Gaussian distributions with all the means as zeros, and all the standard deviations as ones. The parameterized distribution from the encoder is a distribution of the same dimension with parametrized means/standard deviation. Minimizing the Kullback-Leibler (KL) divergence between the parametrized distribution $Q_\phi(z|x)$ and $\mathcal{N}(0, \mathbf{1})$ encourages the $Q_\phi(z|x)$ to be statistically closer to the Gaussian distribution prior $p(z) = \mathcal{N}(0, \mathbf{1})$. The KL divergence between the encoded latent distribution (approximated posterior) and the prior has a nice analytical form for Gaussian distributions with a diagonal covariance matrix (you can find the derivation here):

$$\begin{aligned} L_{regularization} &= KL(Q_\phi(z|x)|p(z)) \\ &= KL\left(\mathcal{N}(\mu_\phi(x_i), \sigma^2_\phi(x_i))|\mathcal{N}(0,1)\right) \\ &= \frac{1}{N_{batch}} \sum_i^{N_{batch}} \frac{1}{2} \left(\sum_d^{N_z} \sigma_{d,\phi}(x_i)^2 + \mu_{d,\phi}(x_i)^2 - \log \sigma_{d,\phi}(x_i)^2 - 1 \right) \end{aligned} \tag{7}$$

$d \in \{1, ..., N_z\}$ is the index for the latent dimension.

For this problem, you need to implement the reconstruction (equation 6) loss and the KL divergence (equation 7) in the `loss_function()` function.

Part 2.4 (5 points) Train your model

After implementing the reparameterization trick and loss function, you should be able to train a model with the train and test loop we provided to you. Load the pre-trained model with the code we provided to you. Train the VAE for 50 epochs. Make sure you obtain a training and test loss below 0.15 before proceeding to the next part. We recommend you save the trained model periodically in your Google Drive. Choose $\beta = 0.001$. This will take around 15 minutes, but shouldn't take too long.

Part 2.5 (20 points) Sample new molecules

The latent space learned by the model is an learned continuous space which you can navigate. The space encodes the complicated molecular 'grammar' of the data it trained on. By sampling vectors z, you can then use a decoder to reconstruct the continuous representation back to a SMILES string. Now use your trained model to sample novel molecules from the learned distribution. Randomly select two SMILES in your test data, and encode into latent vectors with the encoder and then linearly interpolate between the two molecules in the latent space to obtain 10 points in the z space. For each sampled latent code, decode z back to SMILES strings. You can use the `index2SMILES()` function we provided to convert categorical values to SMILES strings. We provide a function `check_SMILES()` to check the validity of generated SMILES strings.

Finally, produce a scatter plot with the first two dimensions of z of your test molecules and newly sampled molecules in the same figure, label your generated SMILES with the decoded SMILES in the legend. Among the 10 molecules you sampled, how many were decoded into a valid SMILES string? (Don't worry if most of the molecules don't decode into valid SMILES strings.) Use RDKit to show 2D line drawings of valid SMILES you generated. Can you propose a reason for why your VAE sometimes fails to generate valid SMILES strings?

References

[1] Gómez-Bombarelli, R. *et al.* Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science* **4**, 268–276 (2018).

[2] Figurnov, M., Mohamed, S. & Mnih, A. Implicit reparameterization gradients. *arXiv preprint arXiv:1805.08498* (2018).

[3] Higgins, I. *et al.* β-VAE: Learning basic visual concepts with a constrained variational framework (2016).