6.047 Problem Set 4 Writeup

Matthew Feng

November 5, 2018

1 Simulated GWAS

(a) β_i and SNP odds ratio

The SNP's odd ratio (OR) is defined as follows:

$$OR = \frac{\frac{\mathbb{P}[y > 2 \mid x_i = 1]}{\mathbb{P}[y < 2 \mid x_i = 1]}}{\frac{\mathbb{P}[y > 2 \mid x_i = 0]}{\mathbb{P}[y < 2 \mid x_i = 0]}}$$

Assume that we only have one SNP to look at, our model becomes $y = \beta x + \epsilon$. We can then make the following substitutions:

$$OR = \frac{\frac{\mathbb{P}[\beta + \epsilon > 2 \mid x_i = 1]}{\mathbb{P}[\beta + \epsilon < 2 \mid x_i = 1]}}{\frac{\mathbb{P}[\epsilon > 2 \mid x_i = 0]}{\mathbb{P}[\epsilon < 2 \mid x_i = 0]}} \tag{1}$$

$$= \frac{\frac{1 - \Phi(2 - \beta)}{\Phi(2 - \beta)}}{\frac{1 - \Phi(2)}{\Phi(2)}}. \tag{2}$$

(b), (c)

See `problem1/simulated_gwas.py`.

(d) Bonferroni Correction

If we are using a significance level of $\alpha = 0.05$, then the p-value needed for significance according to the Bonferroni correction is $\frac{0.05}{m}$. The Bonferroni correction is necessary when performing GWAS because we are conducting multiple significance tests simultaneously; if we continued to only use α, then by chance alone $m \times 0.05$ SNPs would be statistically significant at the $\alpha = 0.05$ level.

(e) Hyperparameters

Accuracy (AC) ($\frac{TP+TN}{TP+TN+FP+FN}$)

Precision (PRC) ($\frac{TP}{TP+FP}$)

Recall (RCL) ($\frac{TP}{TP+FN}$)

Hyperparameters	**TP**	**FP**	**TN**	**FN**	**AC**	**PRC**	**RCL**
$n = 10000, k = 100$ $m = 1000, s = 0.25$	8	0	900	92	0.908	1.00	0.08
$n = 1000, k = 100$ $m = 1000, s = 0.25$	0	1	899	100	0.899	0.00	0.00
$n = 100000, k = 100$ $m = 1000, s = 0.25$	49	0	900	51	0.959	1.00	0.49
$n = 10000, k = 100$ $m = 10000, s = 0.25$	8	0	9900	92	0.991	1.00	0.08
$n = 10000, k = 100$ $m = 1000, s = 0.5$	10	0	900	90	0.910	1.00	0.10
$n = 10000, k = 100$ $m = 1000, s = 0.1$	3	0	900	97	0.903	1.00	0.03

The Bonferroni correction is conservative in categorizing SNPs as contributors to the "affected" phenotype; that is, recall tends to be very low (across all hyperparameter settings), while precision is high (typically 1.00).

As the number of samples increases (i.e. the number of people n), accuracy, precision, and recall all improve. As the standard deviation s of the disease related SNPs increases, so too does recall. When the number of SNPs m increases, accuracy increases as the model tends to correctly classify negatives, but recall remains low.

2 Finding eQTLs

(a) Principal Components Analysis

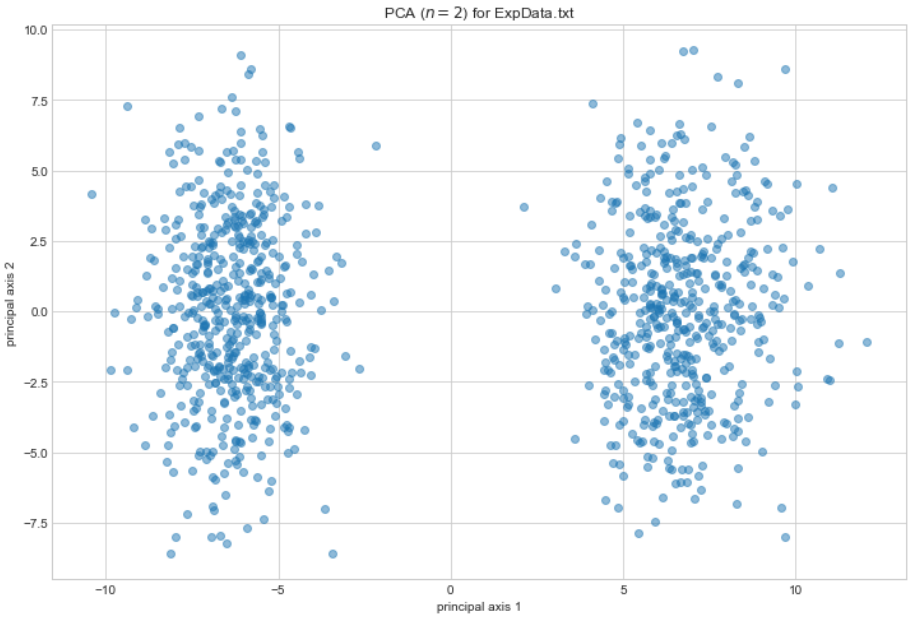

Figure 1: Scatterplot of PCA of the first two principal axes.

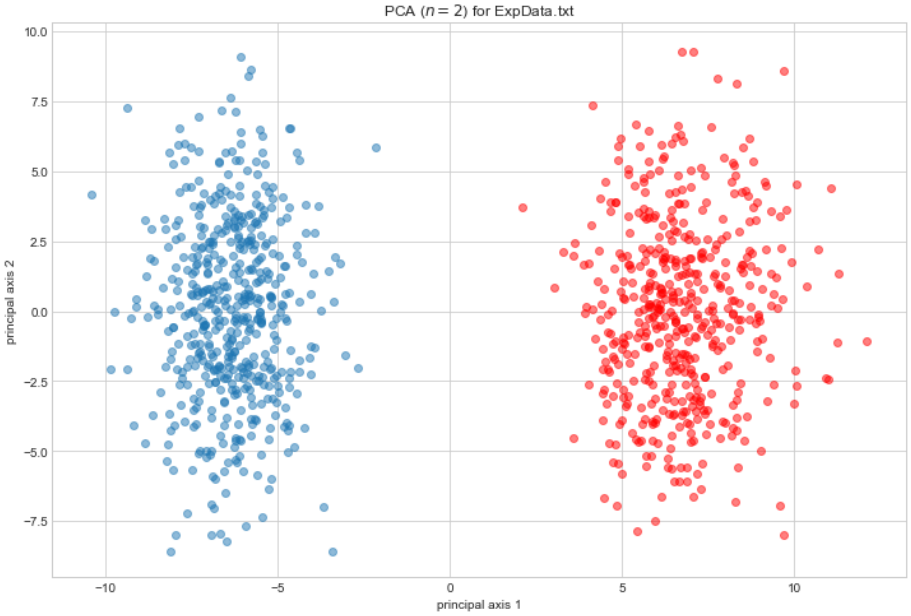

Figure 2: Scatterplot of PCA of the first two principal axes, colored by cluster using the k-means algorithm.

The structure inherent in this population is two clusters, likely those who express certain genes and therefore have some phenotype and those who do not express those genes to a high enough degree for phenotypic visibility. The first principal axis captures this variability; the remaining axes reveal a Gaussian-like distribution within the two clusters, which can cover the range of phenotypic expression.

The code used to create the graphs and perform principal component analysis and k-means clustering is in `problem2/PCA Analysis.ipynb`.

(b) Finding eQTLs via Linear Regression

For every SNP x_i, we find the mean and variance (μ_i, σ_i^2) of the correlation coefficients r_{ij} that x_i has with expression of gene y_j. In this way, we are determining the "typical" contribution of SNP x_i to any gene. We then select the SNP and gene pairs (x_i, e_j) for which r_{ij} is satistically significant under the null hypothesis $H_0 : \rho_{ij} = \mu_i$ (i.e. contribution of x_i to y_j is typical).

To implement the Bonferroni correction, we test each hypothesis that SNP x_i contributes to the expression of gene y_j with significance of $\alpha/(2500000)$, where $\alpha = 0.0001$ (since we are testing 5000 different genes and 500 SNPs, which corresponds with 2500000 different hypotheses).

SNP	Genes
SNP_119	Gene_AIU28, Gene_XMY78, Gene_PDA21, Gene_LKJ65, Gene_TXM58, Gene_BAF02, Gene_HXS05, Gene_FZA89, Gene_TYZ45, Gene_UUH92, Gene_FZL13, Gene_LOE40, Gene_WFM86, Gene_LQE83, Gene_EVL77, Gene_AGR67, Gene_UDL16, Gene_MQT38, Gene_FVH87, Gene_CLI02, Gene_YSN10, Gene_MEN85, Gene_UJU38, Gene_AOG34, Gene_MST69, Gene_ZUG63, Gene_NUY56, Gene_WNN25, Gene_WOZ21, Gene_OHM64, Gene_TAY33, Gene_JNZ42
SNP_157	Gene_AIU28, Gene_XMY78, Gene_PDA21, Gene_LKJ65, Gene_TXM58, Gene_BAF02, Gene_HXS05, Gene_FZA89, Gene_TYZ45, Gene_UUH92, Gene_FZL13, Gene_LOE40, Gene_WFM86, Gene_LQE83, Gene_EVL77, Gene_AGR67, Gene_UDL16, Gene_MQT38, Gene_FVH87, Gene_CLI02, Gene_YSN10, Gene_MEN85, Gene_UJU38, Gene_AOG34, Gene_MST69, Gene_ZUG63, Gene_NUY56, Gene_WNN25, Gene_WOZ21, Gene_OHM64, Gene_TAY33, Gene_JNZ42
SNP_473	Gene_AIU28, Gene_XMY78, Gene_PDA21, Gene_LKJ65, Gene_TXM58, Gene_BAF02, Gene_HXS05, Gene_FZA89, Gene_TYZ45, Gene_UUH92, Gene_FZL13, Gene_LOE40, Gene_WFM86, Gene_LQE83, Gene_EVL77, Gene_AGR67, Gene_UDL16, Gene_MQT38, Gene_FVH87, Gene_CLI02, Gene_YSN10, Gene_MEN85, Gene_UJU38, Gene_AOG34, Gene_MST69, Gene_ZUG63, Gene_WNN25, Gene_WOZ21, Gene_OHM64, Gene_TAY33, Gene_JNZ42

These are likely to be eQTLs because they correlate (based on their R^2 statistic) with a large number of genes at a very strict significance level ($\alpha = 0.00001$, with an additional correction factor of 2500000). The Bonferroni correction tends to be conservative in attributing significance as well, so this increases our belief that the SNPs selected are eQTLs.

Below are the plots of the Student's t-distribution with d.f. = 998 and the locations of the R^2 value for a particular gene for each of the three SNPs identified as eQTLs using the procedure described above.

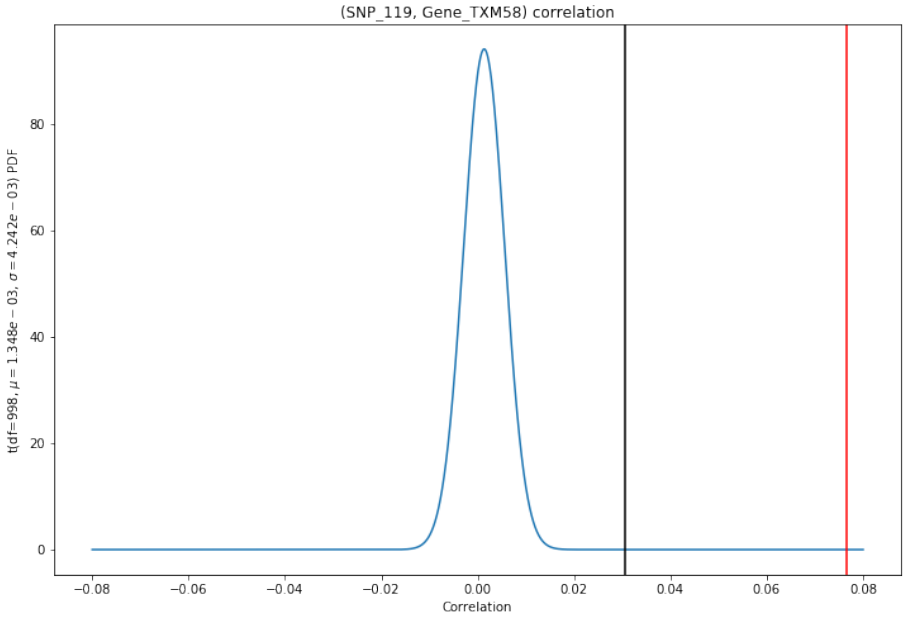

Figure 3: SNP_119 and Gene_TXM58 correlation. Red is the R^2 correlation coefficient between the SNP and the gene, and the black is the critical value above which the R^2 value is significant, using $\alpha = 0.00001$ with a correction factor of 2500000.

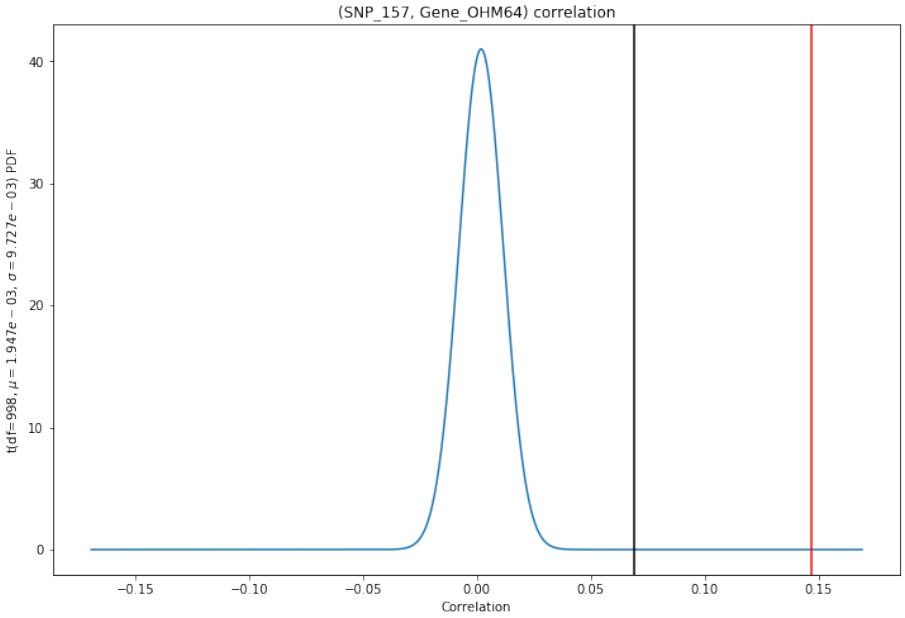

Figure 4: SNP_157 and Gene_OHM64 correlation. Red is the R^2 correlation coefficient between the SNP and the gene, and the black is the critical value above which the R^2 value is significant, using $\alpha = 0.00001$ with a correction factor of 2500000.

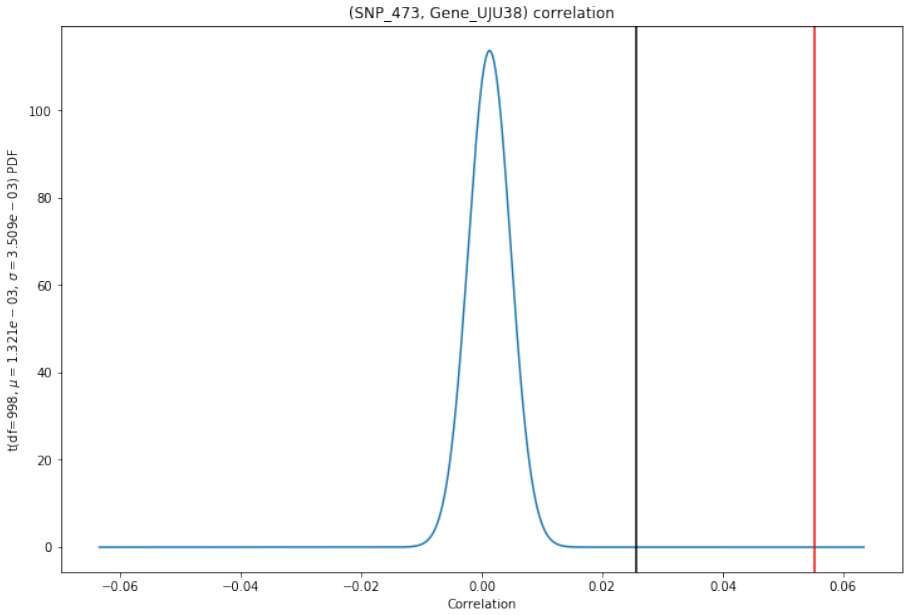

Figure 5: SNP_473 and Gene_UJU38 correlation. Red is the R^2 correlation coefficient between the SNP and the gene, and the black is the critical value above which the R^2 value is significant, using $\alpha = 0.00001$ with a correction factor of 2500000.

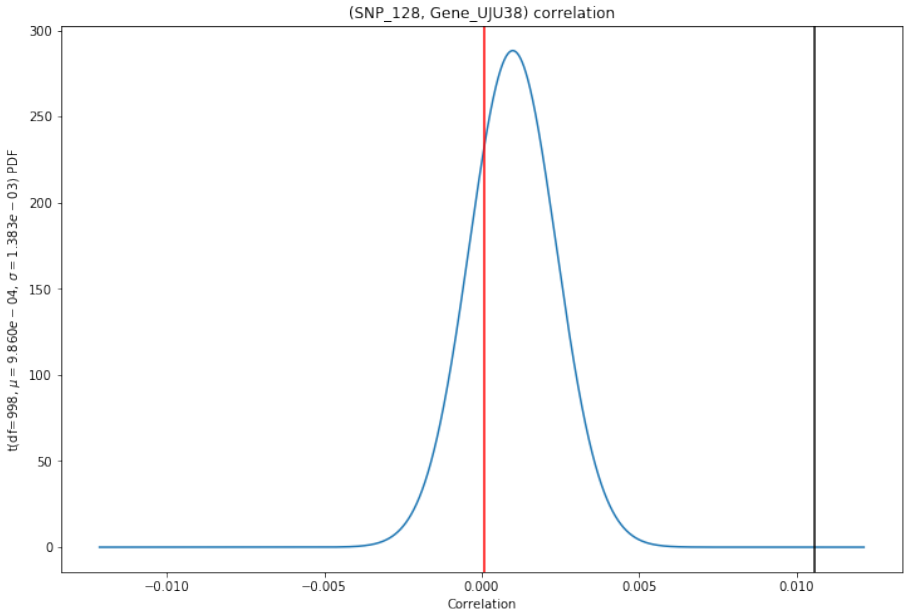

Figure 6: SNP_128 and Gene_UJU38 correlation. This is an example of a correlation that is insignificant. Red is the R^2 correlation coefficient between the SNP and the gene, and the black is the critical value above which the R^2 value is significant, using $\alpha = 0.00001$ with a correction factor of 2500000.

The code used to compute the R^2 values and distribute the computation across multiple AWS instances is in `problem2/gene_snp.py`.

The code used to find which R^2 values were significant is in the Python notebook `problem2/Finding Significant SNP-Gene pairs.ipynb`.

(c) Additional Datasets

Two datasets that would have been useful in constraining the number of (SNP, Gene) pairs that had to be considered are the following:

Minor Allele Frequencies This dataset would tell us what the minor allele frequency for each SNP would be (i.e. columns=SNPs, row=MAF). This is helpful because we can immediately exclude SNPs with a MAF $< 5\%$, as those would requires very strong statistical power to make meaningful statements.

Hardy-Weinberg Equilibrium Filter. This dataset (i.e. columns=SNPs, rows=Hardy-Weinberg equilibrium proportions) would allow us to discard SNPs that violate the Hardy-Weinberg assumptions, meaning that it is likely that there is a genotyping error or that a certain allele is being selected for (or against), and thus should not be used.

3 Convolutional Neural Networks

(a) Implementation

```
#!/usr/bin/env python

from keras.models import *
from keras.layers import *
import keras

import numpy as np

from datetime import datetime

import alternative_models as models

import argparse

BATCH_SIZE = 10
NUM_EPOCHS = 20
KERNEL_SIZE = (4, 4)
POOL_SIZE = (4, 6)
HIDDEN_UNITS = 32
CONV_FILTERS = 32
MODEL_NAME = None

def get_x_y_data():
    negative_data = []
    with open('negativedata.txt') as f:
        for line in f:
            final_mat = np.zeros((4,len(line)-1,1))
            for i in range(len(line)):
                char = line[i]
                if char == 'a':
                    final_mat[:,i,:] = np.array([[1],[0],[0],[0]])
                if char == 'c':
                    final_mat[:,i,:] = np.array([[0],[1],[0],[0]])
                if char == 'g':
                    final_mat[:,i,:] = np.array([[0],[0],[1],[0]])
                if char == 't':
                    final_mat[:,i,:] = np.array([[0],[0],[0],[1]])
            negative_data.append(final_mat)

    positive_data = []
    with open('positivedata.txt') as f:
```

```
        for line in f:
            final_mat = np.zeros((4,len(line)-1,1))
            for i in range(len(line)):
                char = line[i]
                if char == 'a':
                    final_mat[:,i,:] = np.array([[1],[0],[0],[0]])
                if char == 'c':
                    final_mat[:,i,:] = np.array([[0],[1],[0],[0]])
                if char == 'g':
                    final_mat[:,i,:] = np.array([[0],[0],[1],[0]])
                if char == 't':
                    final_mat[:,i,:] = np.array([[0],[0],[0],[1]])
            positive_data.append(final_mat)

    X = np.array(negative_data + positive_data)
    y = np.array([0] * len(negative_data) + [1] * len(positive_data))
    y = keras.utils.to_categorical(y)

    X_neg = X[:len(negative_data), ...]
    X_pos = X[len(negative_data):, ...]
    y_neg = y[:len(negative_data), ...]
    y_pos = y[len(negative_data):, ...]

    return X_neg, X_pos, y_neg, y_pos

def create_model():
    model = Sequential()
    model.add(Conv2D(CONV_FILTERS,
                     input_shape=(4, 100, 1),
                     kernel_size=KERNEL_SIZE,
                     activation="relu",
                     padding="same"))
    model.add(MaxPool2D(pool_size=POOL_SIZE))
    model.add(Flatten())
    model.add(Dense(HIDDEN_UNITS, activation="relu"))
    model.add(Dense(2, activation="softmax")) # same as 1 output sigmoid
    return model

MODEL_FUNC = create_model

def main():
    np.random.seed(1)

    TRAIN_TEST_FRAC = 0.9
    DATASET_SIZE = 5000
    # 10000 x (4, 100, 1) images total (5000 examples each)
```

```
    SPLIT = int(TRAIN_TEST_FRAC * DATASET_SIZE)

    Xn, Xp, yn, yp = get_x_y_data()
    shuffled_order = np.arange(0, DATASET_SIZE)
    np.random.shuffle(shuffled_order)
    Xn, Xp = Xn[shuffled_order, ...], Xp[shuffled_order, ...]
    yn, yp = yn[shuffled_order, ...], yp[shuffled_order, ...]

    X_train = np.vstack((Xn[:SPLIT, ...], Xp[:SPLIT, ...]))
    y_train = np.vstack((yn[:SPLIT, ...], yp[:SPLIT, ...]))

    X_test = np.vstack((Xn[SPLIT:, ...], Xp[SPLIT:, ...]))
    y_test = np.vstack((yn[SPLIT:, ...], yp[SPLIT:, ...]))

    print(X_train.shape)

    # define model
    model = MODEL_FUNC()
    model.compile(loss="categorical_crossentropy",
        optimizer="adam",
        metrics=["accuracy"])

    start = datetime.now()

    if MODEL_NAME == "lstm":
        X_train = X_train.squeeze()
        X_train = np.swapaxes(X_train, 1, 2)
        X_test = X_test.squeeze()
        X_test = np.swapaxes(X_test, 1, 2)

    model.fit(X_train, y_train, epochs=NUM_EPOCHS, batch_size=BATCH_SIZE)
    end = datetime.now()

    scores = model.evaluate(X_test, y_test)

    print("\n{}: {:.2f}%".format(model.metrics_names[1], scores[1] * 100))
    print("elapsed: {}".format(str(end - start)))

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-b", "--batch_size",
        help="Number of examples per batch",
        type=int)
    parser.add_argument(
        "-e", "--epochs",
```

```
    help="Number of epochs to train over",
    type=int)
parser.add_argument(
    "-k", "--kernel",
    help="height,width tuple representing size of kernel.",
    type=str)
parser.add_argument(
    "-p", "--pool",
    help="height,width tuple representing size of pool.",
    type=str)
parser.add_argument(
    "-u", "--hidden_units",
    help="Number of hidden units for the Dense layer",
    type=int)
parser.add_argument(
    "-f", "--num_filters",
    help="Number of filters for the Conv layer",
    type=int)
parser.add_argument(
    "-m", "--model",
    help="Use a particular model implemented in alternative_models.py",
    type=str)

args = parser.parse_args()

if args.batch_size:
    BATCH_SIZE = args.batch_size

if args.epochs:
    NUM_EPOCHS = args.epochs

if args.kernel:
    height, width = map(int, args.kernel.split(","))
    KERNEL_SIZE = (height, width)

if args.pool:
    height, width = map(int, args.pool.split(","))
    POOL_SIZE = (height, width)

if args.hidden_units:
    HIDDEN_UNITS = args.hidden_units

if args.num_filters:
    CONV_FILTERS = args.num_filters

if args.model:
```

```
        MODEL_NAME = args.model
        MODEL_FUNC = getattr(models, args.model)

    main()

# acc: 94.70%
# elapsed: 0:00:56.455801
```

(b) Layers

Layer	Output Shape	No. of Parameters
Conv2D	(4, 100, 32)	544
MaxPooling2D	(1, 16, 32)	0
Flatten	(512)	0
Dense	(32)	16416
Dense	(2)	66

Convolutional layer. The convolutional layer slides filters (32, in this case) in order to capture local patterns in an image or sequence. Convolution is useful for finding features that provide valuable predictive power for later layers in the network.

Max Pooling layer. The max pooling layer has two main effects: it reduces the dimensionality of the network, allowing it to converge faster and reduce chance of overfitting. At the same time, max pooling also aggregates features that are found at the local level, keeping the most salient to be processed later in the network.

Flatten layer. The Flatten layer reshapes the two dimensional output of the max pooling layer into a single dimensional input that can be fed into the fully connected ("Dense") layers.

Dense(32). The Dense layer, also known as a "fully connected" layer, attempts to take a global consideration of all the lower-level features that were extracted earlier in the network, and outputs a series of higher level features that use the lower-level features as building blocks.

Dense(2). The final Dense layer is the decision layer, outputing whether or not the network classifies the sequence as *positive* or *negative*. This layer could also be a single output layer with "sigmoid" activation (rather than "softmax").

(c) Hyperparameters

Model	Hyperparameters	AC (Train)	AC (Test)	Time
many_epochs	epochs = 300 batch_size = 10 kernel_size = $(4, 4)$ pool_size = $(4, 6)$ hidden_units = 32 num_filters = 32	100.00%	94.80%	00:13:02
many_filters	epochs = 50 batch_size = 10 kernel_size = $(4, 4)$ pool_size = $(4, 6)$ hidden_units = 32 num_filters = 1024	99.32%	95.60%	00:35:37
wide_network	epochs = 50 batch_size = 10 kernel_size = $(4, 4)$ pool_size = $(4, 24)$ hidden_units = 512 num_filters = 32	97.56%	93.50%	00:02:34

Many epochs. This model was trained on 300 epochs, or 300 runs through the training data. As expected, the accuracy on the training data was 100% (fit perfectly), while the accuracy on the test data did not improve significantly (which was surprising, as I had expected a decrease in performance). This signals, however, that the training data and test data are indeed generated from the same distribution, such that even fitting the training data exactly allows predictive power into the test data.

Many filters. The idea behind increasing the number of output filters of the Convolutional layer was driven by the notion that the Convolutional layer provided the more important analysis of the low-level, local patterns that characterized *positive* sequences from *negative* ones. This intuition was validated in the increased performance on the test set.

Wide network. In this third setting of hyperparameters, the size of the Dense layer was expanded to be particularly wide; the idea was to try to allow more higher level features to be formed. However, the test accuracy decreased, likely because the number of lower-level features was not increased, and so the additional width to the Dense layer only increased the capacity of the network to overfit.

(d) Architectures

Model	Architecture	AC (Train)	AC (Test)	Time
`deep_conv_net`	Conv2D(32, (4, 6)) Conv2D(64, (4, 3)) MaxPool((4, 6)) Conv2D(128, (4, 3)) Conv2D(1024, (1, 1)) Dense(64) Dense(2)	99.36%	98.10%	00:10:25
`fully_connected`	Dense(1024) Dense(256) Dense(512) Dense(2)	99.58%	93.20%	00:03:02
`lstm`	LSTM(64) Dense(2)	98.40%	98.30%	00:23:17

Deep convolutional network. Convolutional layers are able to capture low-level features, as well as local structures in the data; stacking convolutional layers allows for higher and more complex patterns to be discovered, while also expanding the size of the patterns that are considered. As such, we see an increased test set accuracy of 98.10%.

Fully connected network. The fully connected network performed suboptimally compared to the other networks because it failed to capture the local structure of sequences in its design. Rather, the fully connected network considered the entire sequence at once, discovering connections that may not exist, or obscuring patterns with the shear number of parameters.

Recurrent network. The image we are feeding into the network is actually a sequence of nucleotides. Recurrent networks were designed to work well on sequential data. In some sense, recurrent networks are like the convolutional layers we were using previously, but are more suited for arbitrary sequences. Particularly with LSTMs, which enables recurrent networks to handle longer sequences, using a recurrent model to handle biological sequences is particular appropriate. This is also apparent in the test accuracy of the model, which is the highest out of all the models at 98.3%.